Al-Azhar University
Faculty of Engineering
Department of Electrical Engineering
Electronics and Communication Engineering

Design and Verification of a Custom RISC-V Vector ISA Extension for AI Acceleration

Presented by:

- Abdelrahman Taha Mohamed
- Ahmed Abdelhameed Mahmoud
- Mohamed Emadeldin Ali Sayed
- Omar Ahmad Abd El-Hamid Amin
- Ramadan Khaled Younis
- Yusuf Mohamed Elsayed

Supervised by:
Dr. Abdallah Mohamed Hassan

Sponsored by:
STMicroelectronics-Egypt

Agenda

01 The Challenge & Our Approach

02 The Vector Extension: Scope & Capabilities

03 Architecture

04 Kernel Design

05 Verification & Validation

06 Results & Benchmarking

07 Conclusion

01

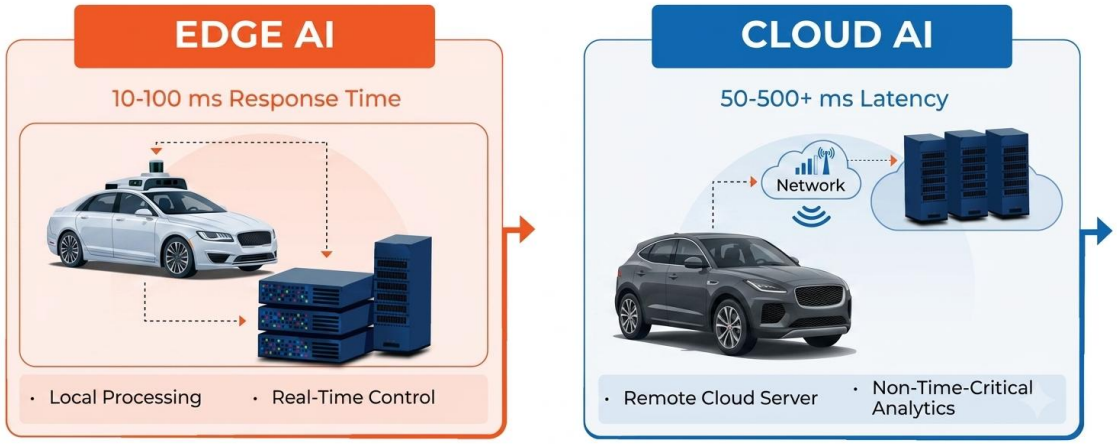
The Challenge & Our Approach

Edge AI needs real-time compute — locally

01

Some decisions are too fast for the cloud.

- A camera in a moving car has to spot a pedestrian and react within a few milliseconds. There is no time to send the frame to a server and wait, and often no reliable connection to send it over.
- So, the model runs on the device itself, where the core is small, the power budget is tight, and the model still expects millions of operations for every frame. That gap is the problem we set out to close.



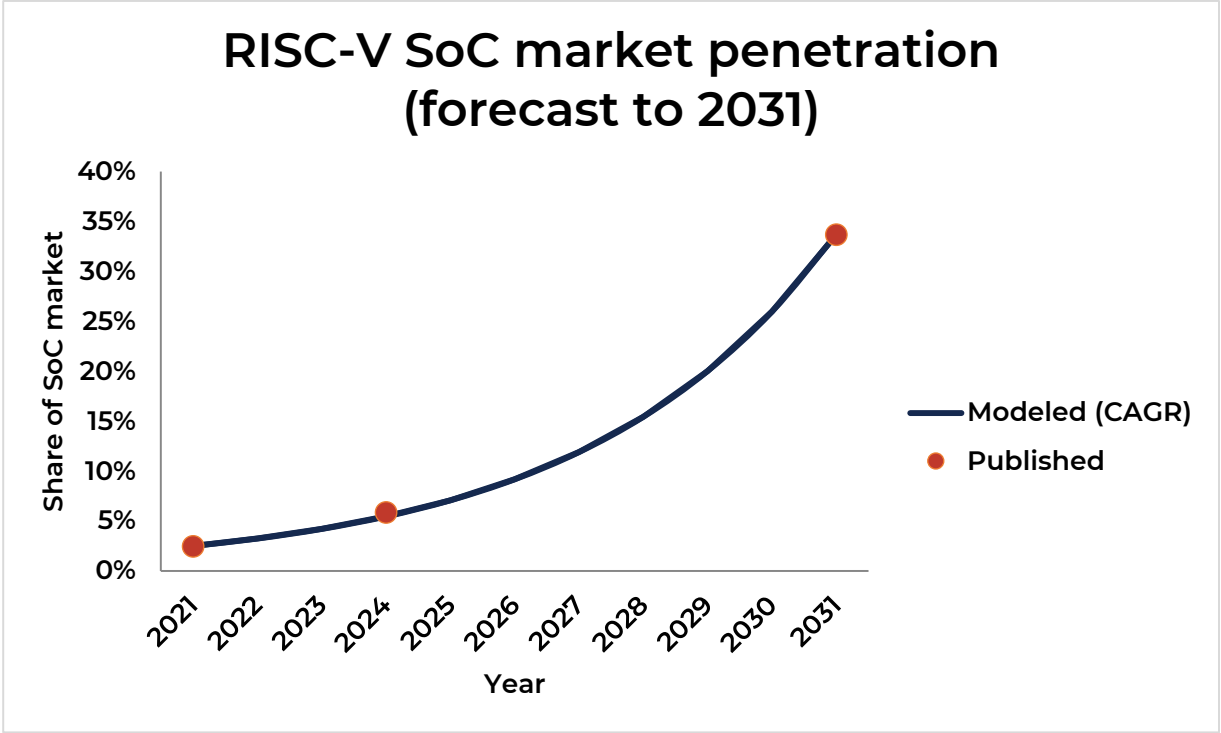
Why RISC-V

Open and royalty-free

- Anyone can use and extend the ISA with no licensing fees or lock-in.

Built to be extended

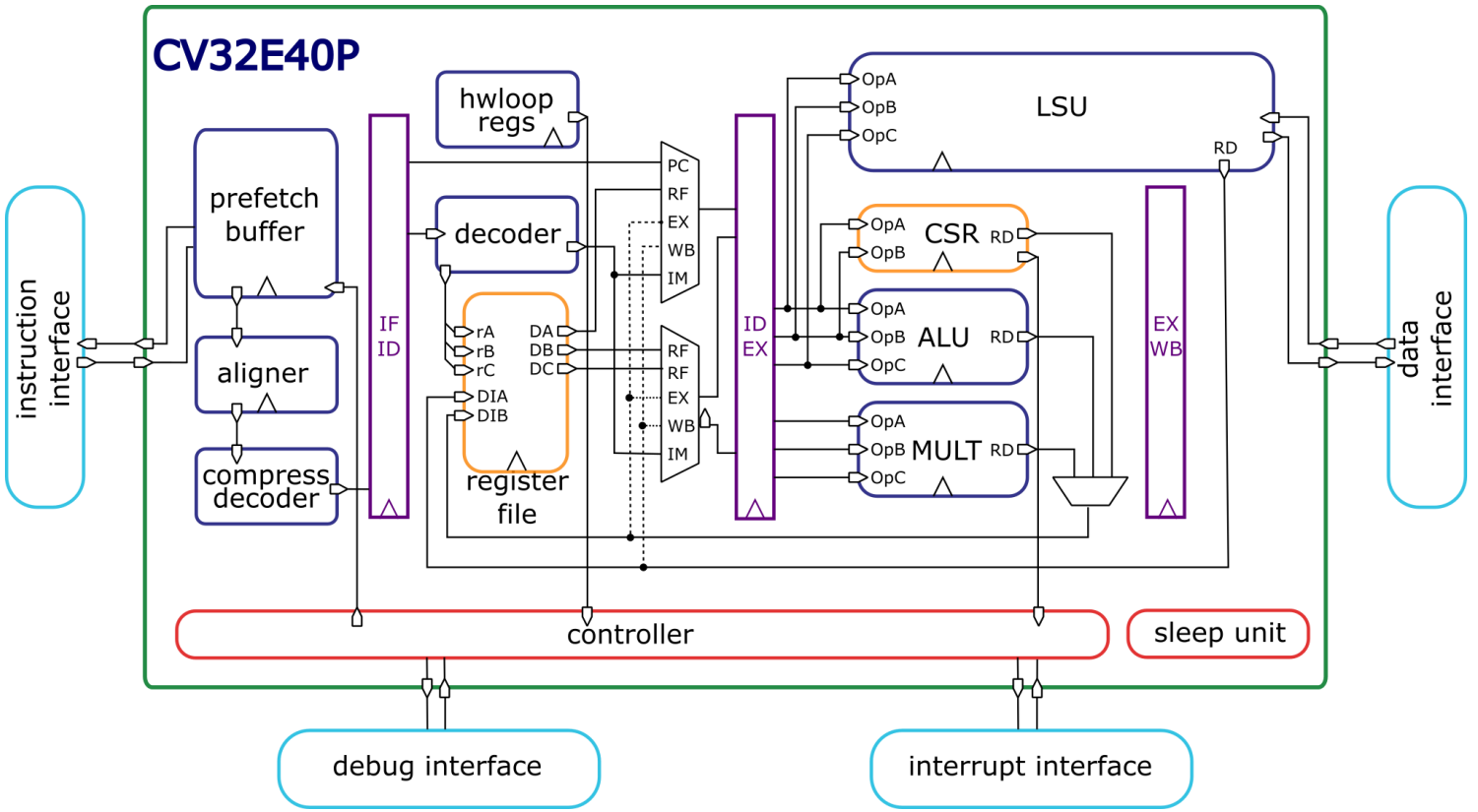
- The standard reserves space for custom instructions, so a vector unit is a supported path, not a workaround.



Why CV32E40P

Silicon-proven, open RTL

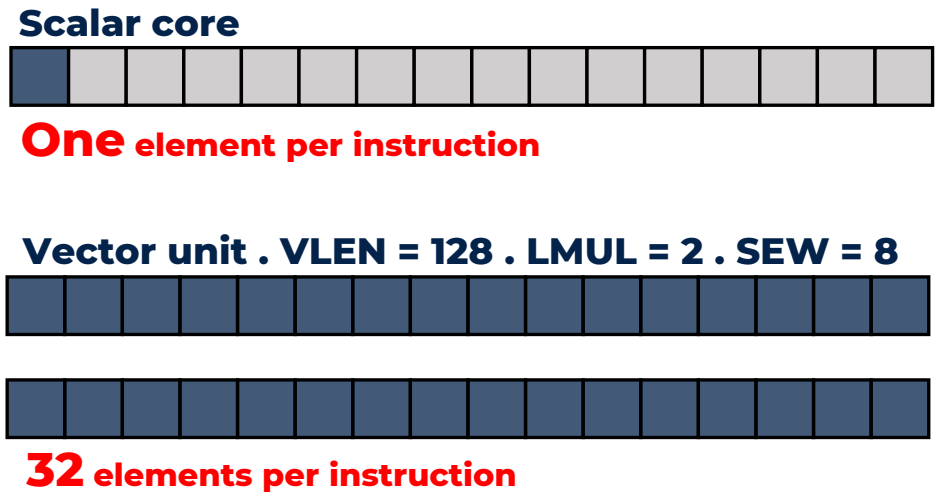
- An open-source 32-bit core from the OpenHW Group — industrially maintained and already verified.



Why a vector extension

AI doesn't need a thousand different operations — it needs the same one, a thousand times. Vectorization is how you stop paying for it a thousand times.

One vector instruction can handle up to 32 elements at once.

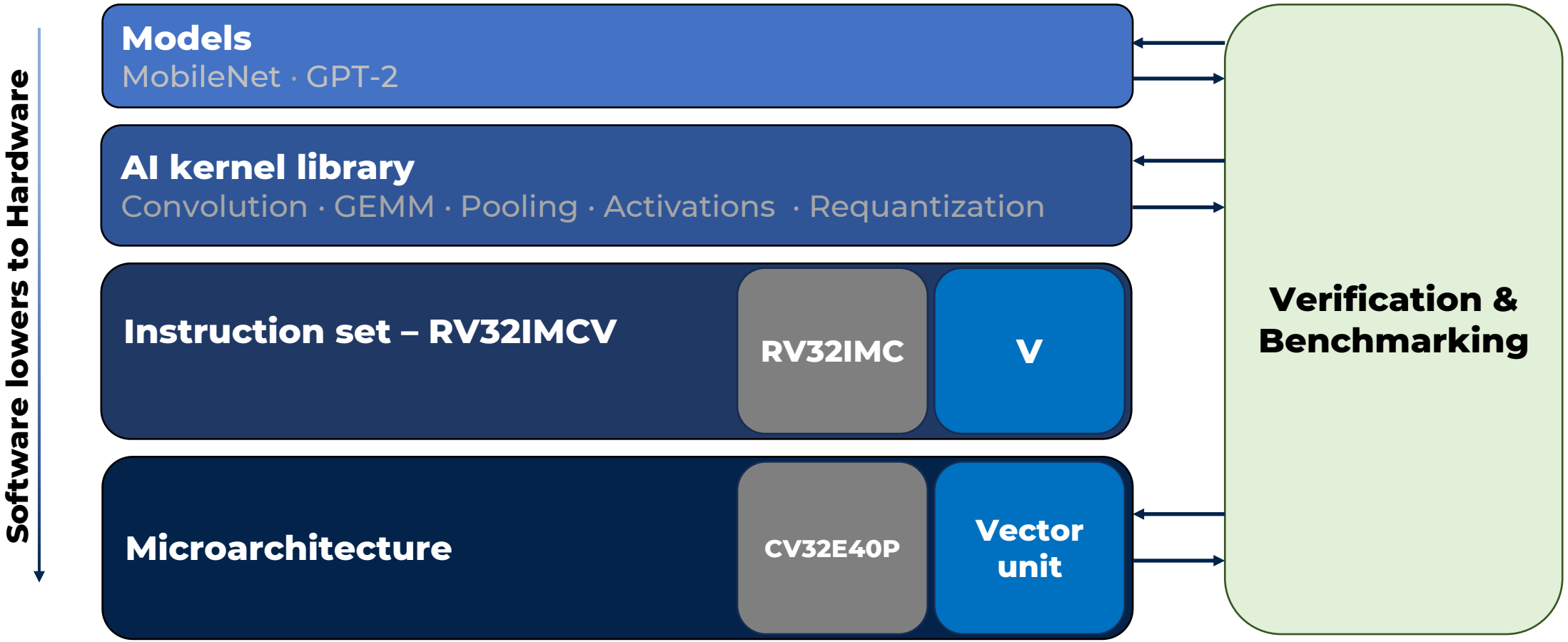


02

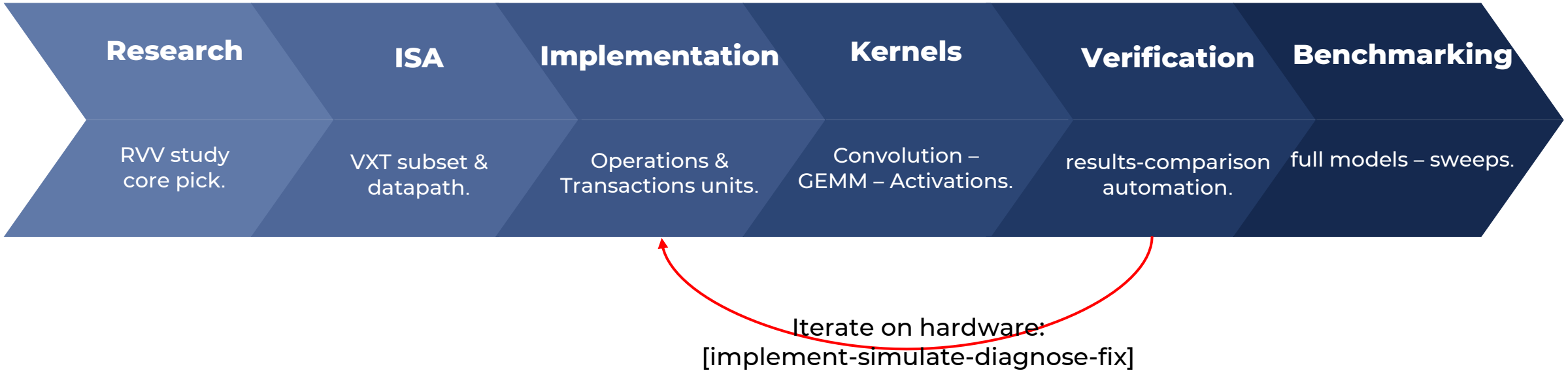
The Vector Extension: Scope & Capabilities

Project scope — the full stack

A custom vector extension, the hardware that runs it, the software that uses it, built on the open CV32E40P core.



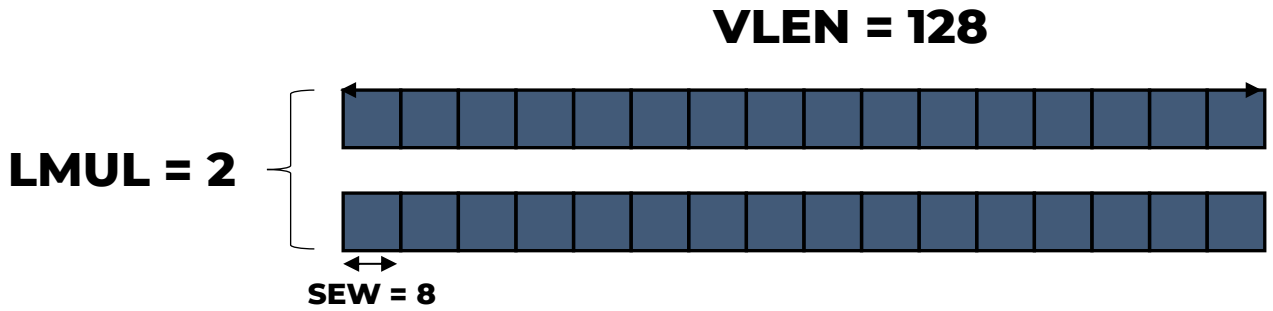
Development phases



Supported features

SEW ∈ {8, 16, 32} — selected element width (widening / narrowing: 8 & 16 only)

LMUL ∈ {1, 2} — group 1 or 2 registers (no fractional LMUL)



Execution — non-pipelined, multi-cycle (one instruction at a time)

SEW	Lanes / Register	with LMUL = 2
8 (int8)	16	32
16 (int16)	8	16
32 (int32)	4	8

Supported instructions

An RVV subset, grouped by function — chosen to cover what AI kernels actually need.

Configuration
vsetvli · csrrw/s/c · csrrwi/si/ci

Memory — load / store
vle.v · vse.v · vlse.v · vsse.v (unit-stride & strided)

Moves & broadcast
vmv.v.x · vmv.v.i · vmv.s.x · vmv.x.s · vmv⟨n⟩r.v

Arithmetic
vadd · vsub · vrsub · vsadd (saturating)

Min / max
vmax · vmaxu · vmin · vminu

Shifts
vsll · vsrl · vsra

Multiply & MAC
vmul · vmulh · vwmul(u) · vwmacc (widening)

Narrowing & clip
vnsrl · vnsra · vnclip

Compare, mask & select
vmseq · vmslt · vmerge · vmnot · vmandn · vid

Reductions
vredsum · vredmax

Scalar vs Vector

Adding 32 int8 elements: c[i] = a[i] + b[i]

Scalar core

```
loop:
  lb   t1, (a0)
  lb   t2, (a1)
  add  t3, t1, t2
  sb   t3, (a2)
  addi a0,a1,a2 ; bnez ...
↻ repeat 32 times
```

~6 instructions per element × 32 — loop overhead every step

Vector unit (vxt)

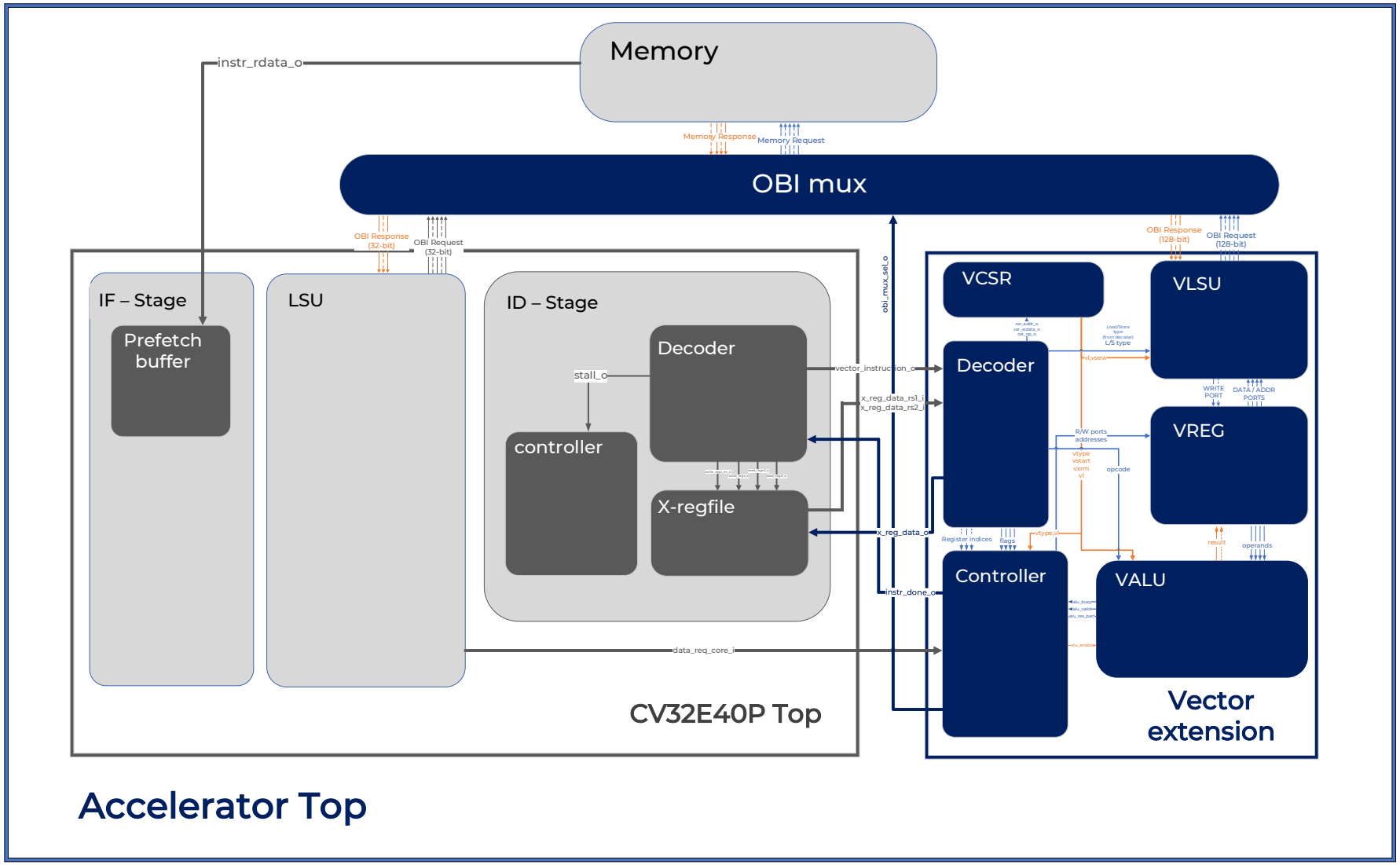
```
vsetvli t0, 32, e8, m2
vle8.v  v0, (a0)
vle8.v  v2, (a1)
vadd.vv v4, v0, v2
vse8.v  v4, (a2)
✓ all 32 elements, one pass
```

5 instructions total — no loop, no per-element overhead

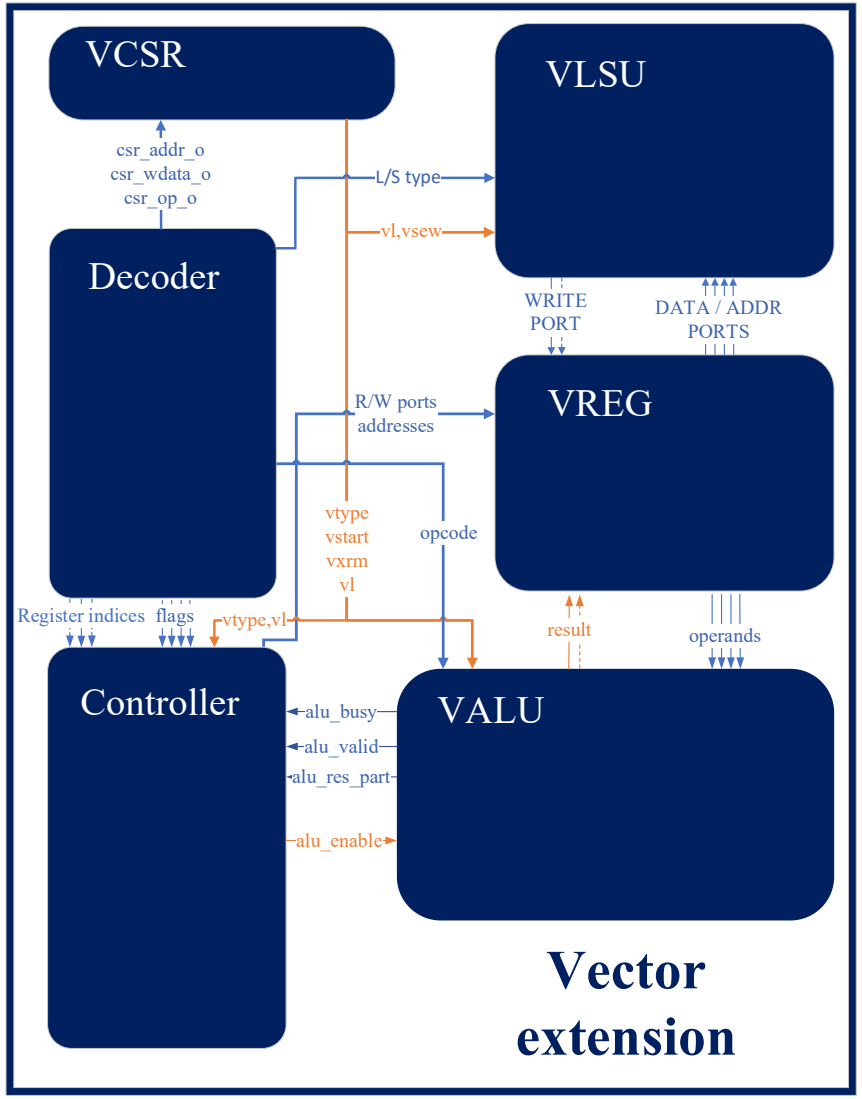
Instructions issued to add 32 int8 element:

03
Architecture

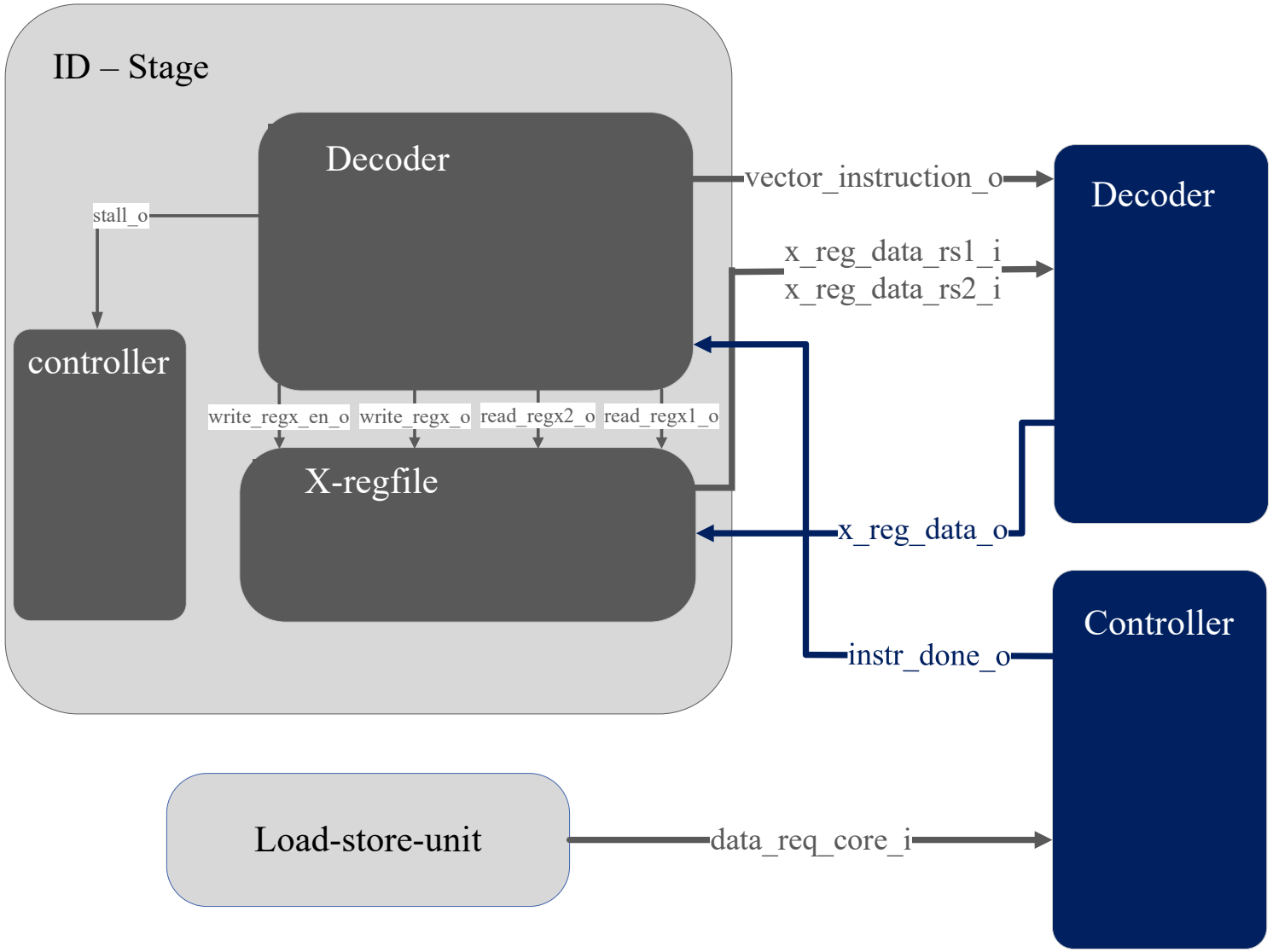
Microarchitecture



Inside the vector extension

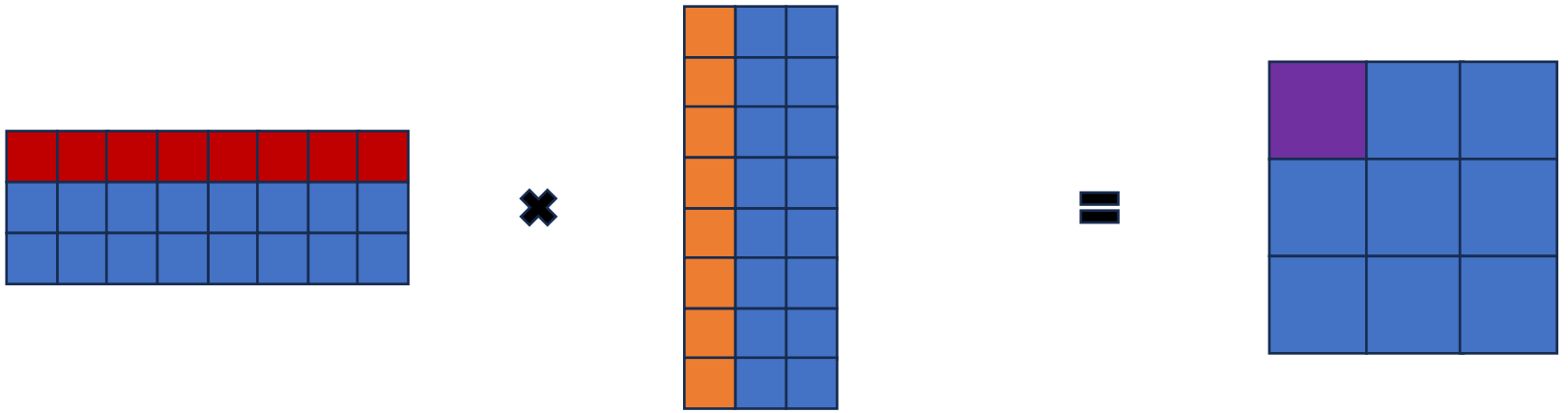


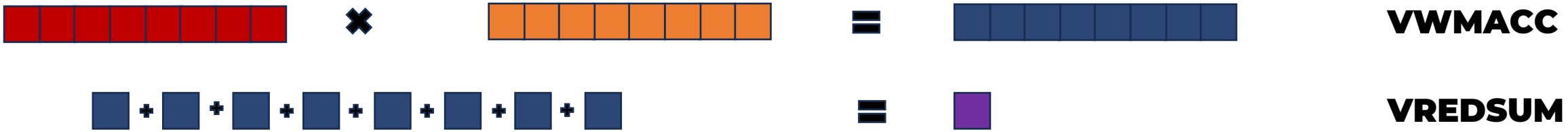
Interface


ID – Stage
Decoder
stall_o
controller
write_regx_en_o
write_regx_o
read_regx2_o
read_regx1_o
X-regfile
vector_instruction_o
x_reg_data_rs1_i
x_reg_data_rs2_i
Decoder
x_reg_data_o
instr_done_o
Controller
Load-store-unit
data_req_core_i

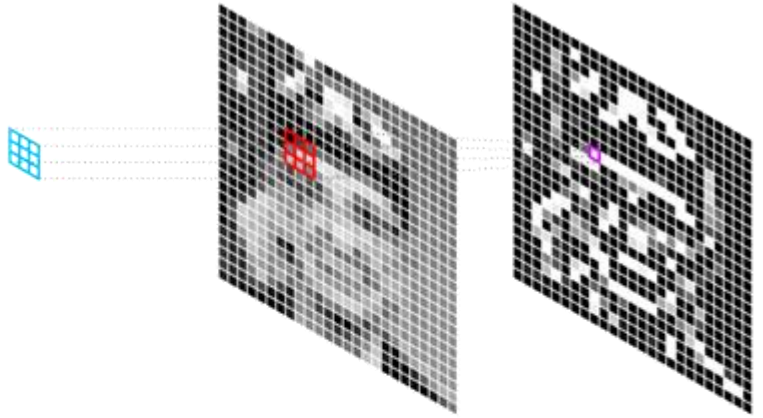
04
Kernel Design

General Matrix Multiplication (GEMM)





Convolution



-1	-1	-1
-1	8	-1
-1	-1	-1

Kernel

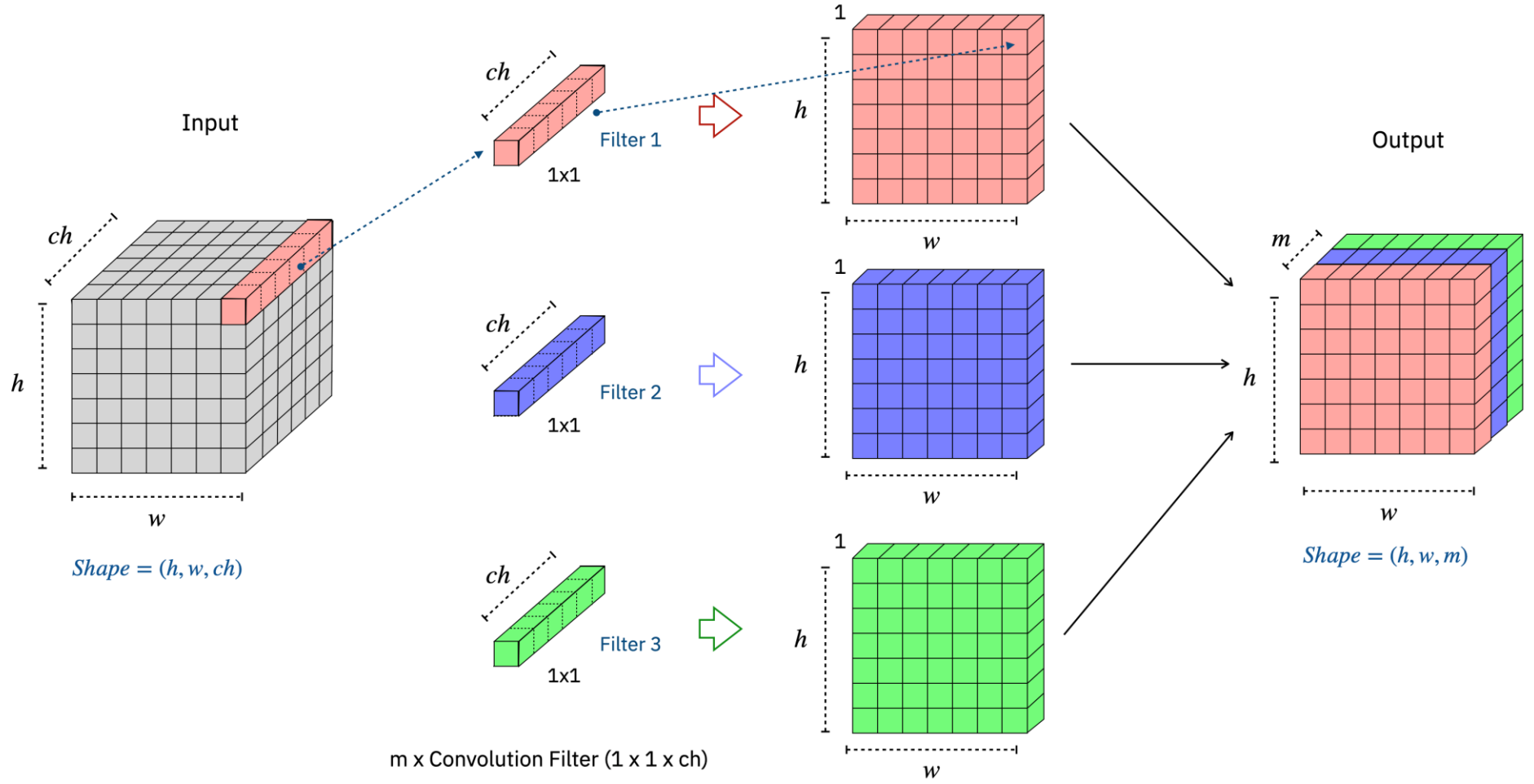
45	81	87
194	203	215
164	116	131

Input

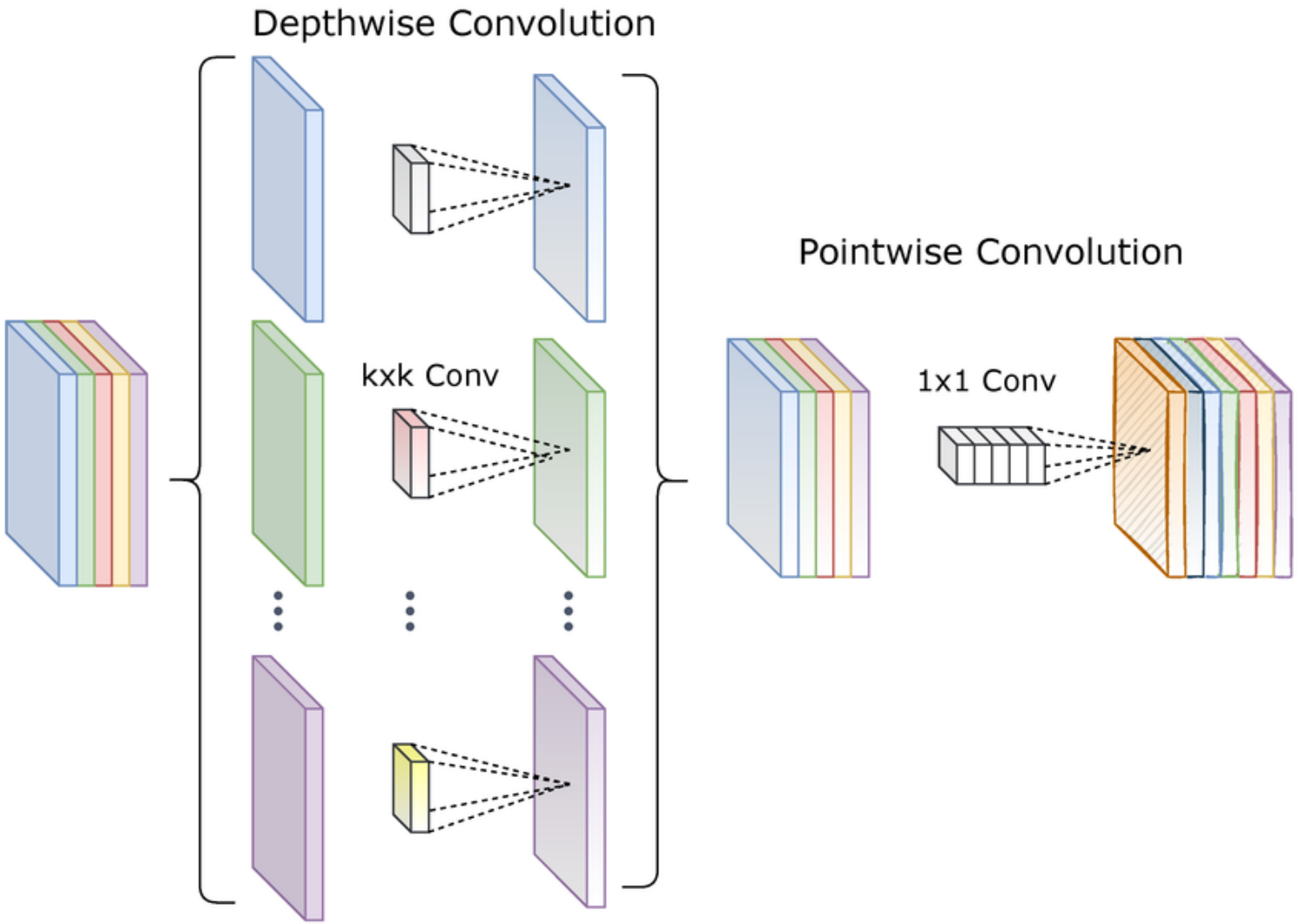
-1	-1	-1	-1	8	-1	-1	-1	-1	✖	45	81	87	194	203	215	164	116	131	=	-45	-81	-87	-194	1624	-215	-164	-116	-131

-45	+	-81	+	-87	+	-194	+	1624	+	-215	+	-164	+	-116	+	-131	=	591

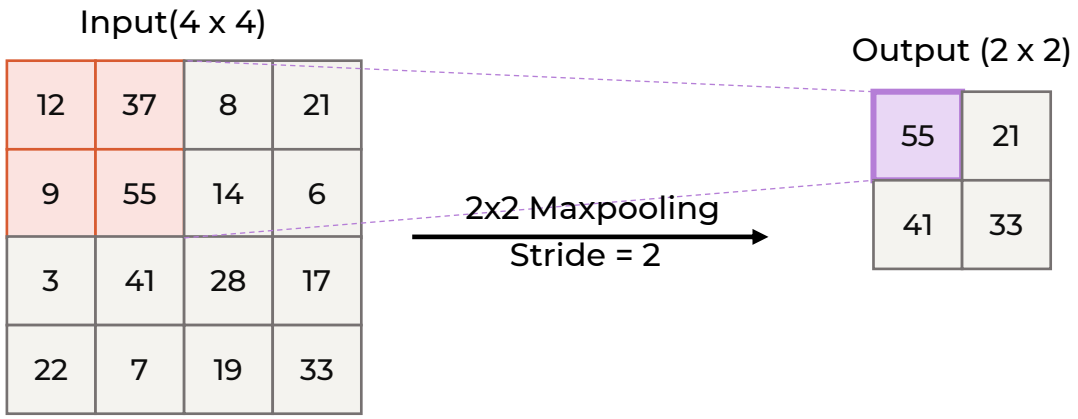
Pointwise Convolution

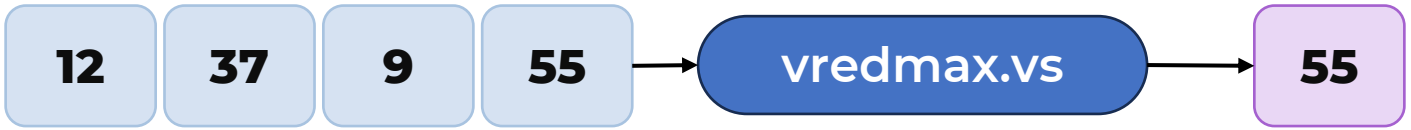


Depthwise Convolution



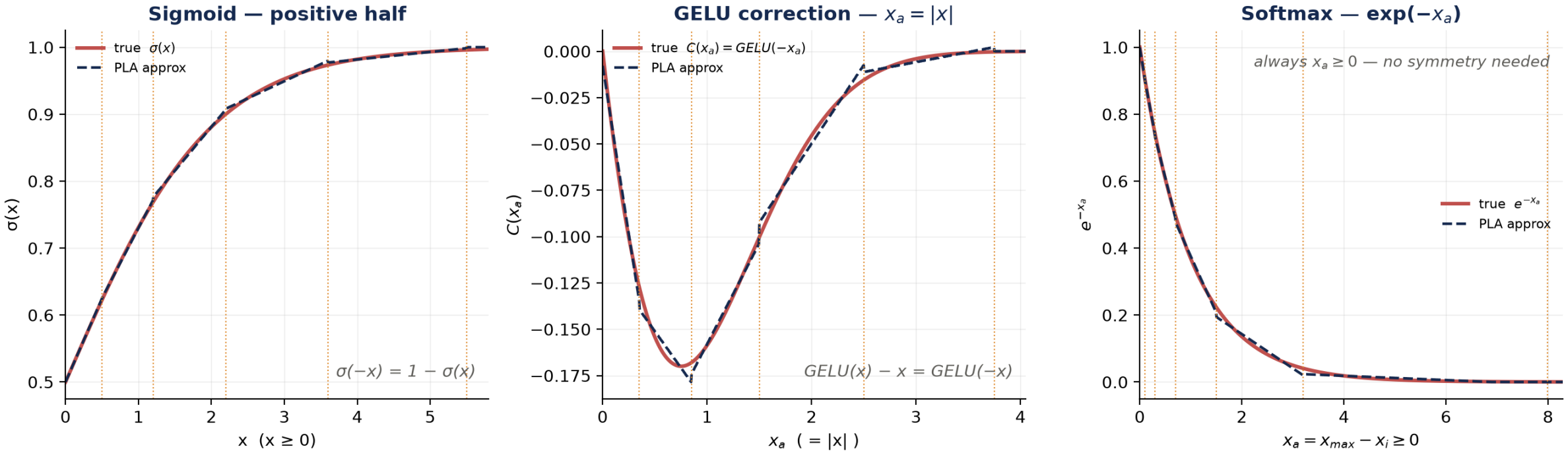
MaxPooling


Input(4 x 4)
12
37
8
21
9
55
14
6
3
41
28
17
22
7
19
33
2x2 Maxpooling
Stride = 2
Output (2 x 2)
55
21
41
33


12
37
9
55
vredmax.vs
55

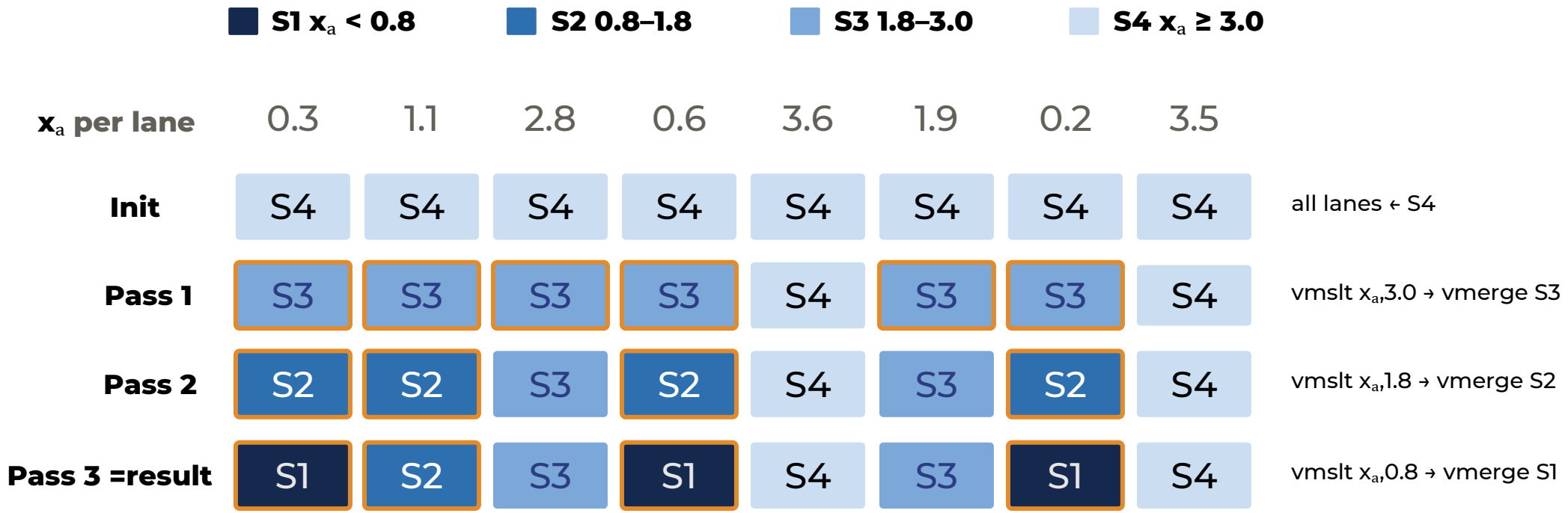
Non-Linear Functions

Piecewise-linear approximation of non-linear activations (Q3.12 fixed-point)



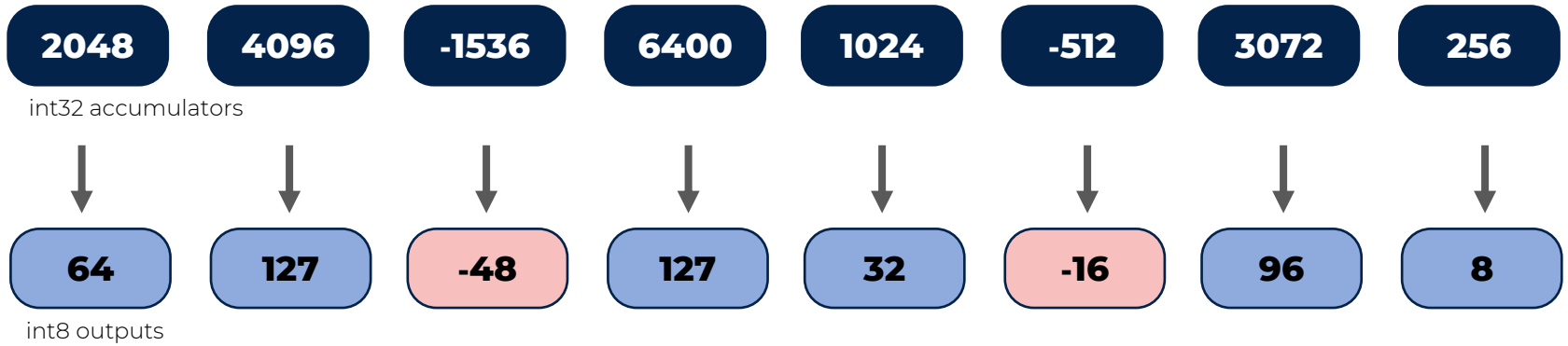
Non-Linear Functions

A vector lane can't branch — so we apply the outer segment to all lanes, then overwrite inner ones with a mask.


S1 x_a < 0.8
S2 0.8–1.8
S3 1.8–3.0
S4 x_a ≥ 3.0
x_a per lane
0.3 1.1 2.8 0.6 3.6 1.9 0.2 3.5
Init
S4 S4 S4 S4 S4 S4 S4 S4
all lanes ← S4
Pass 1
S3 S3 S3 S3 S4 S3 S3 S4
vmslt x_a,3.0 → vmerge S3
Pass 2
S2 S2 S3 S2 S4 S3 S2 S4
vmslt x_a,1.8 → vmerge S2
Pass 3 =result
S1 S2 S3 S1 S4 S3 S1 S4
vmslt x_a,0.8 → vmerge S1

- Each pass overwrites only the masked lanes — the innermost match wins (last write).
- One mechanism, no per-lane branching — the same pattern serves sigmoid, GELU, and softmax's exp.

Requantization



VNCLIP.WI
>> 5 (× 1/32),
clip to int8

05
Verification & Validation

Block-level verification

For every vector extension block, a dedicated SystemVerilog testbench applied directed test cases and compared the output with the expected result.

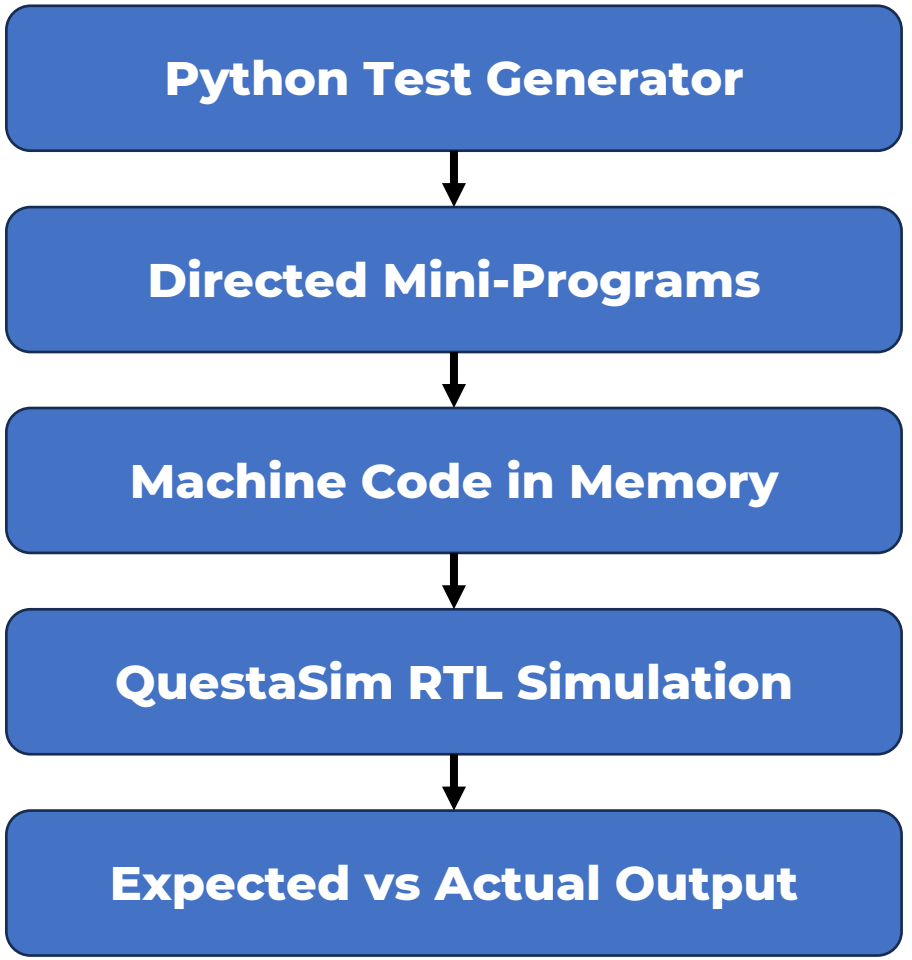
This helped us catch local issues early before connecting the blocks together.

Following internal assembly, the entire integrated vector extension was verified as a standalone subsystem prior to integration with the main core.

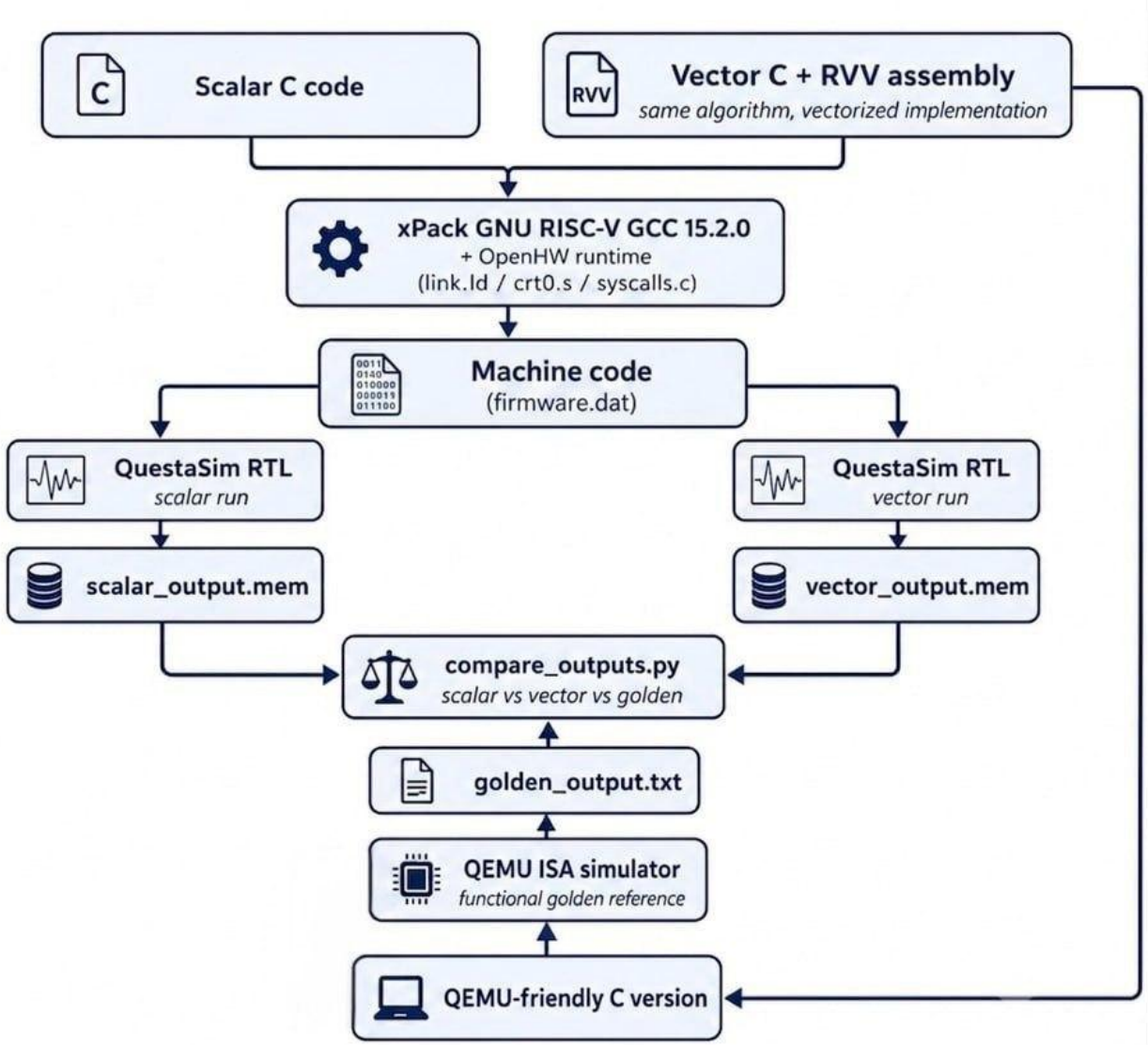
Instruction-Level RTL Verification

Each supported vector instruction was tested individually on the integrated RTL system.

- Python scripts generated directed mini-programs for each supported vector instruction.
- The generated machine-code programs were loaded into system memory and simulated on the complete accelerator system: CV32E40P core + vector extension.
- The output of each instruction was checked against the expected result to confirm functional correctness before kernel-level validation.



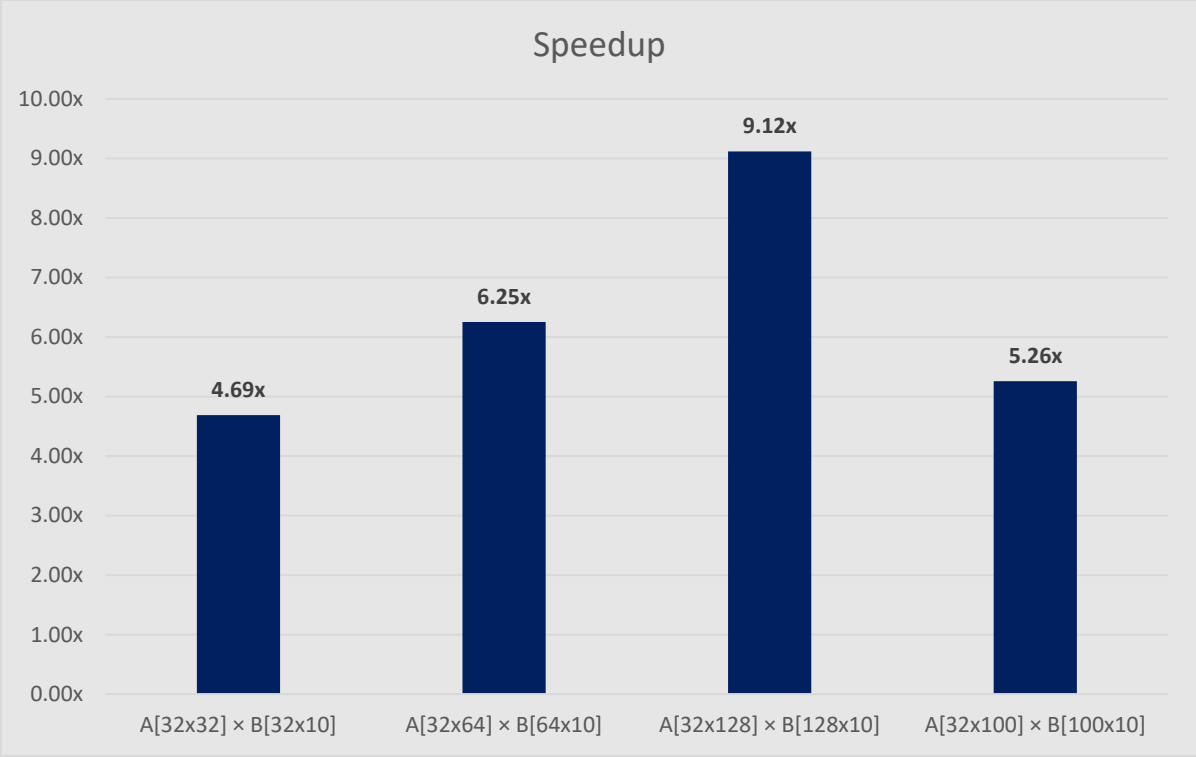
Kernel & Model Verification Methodology

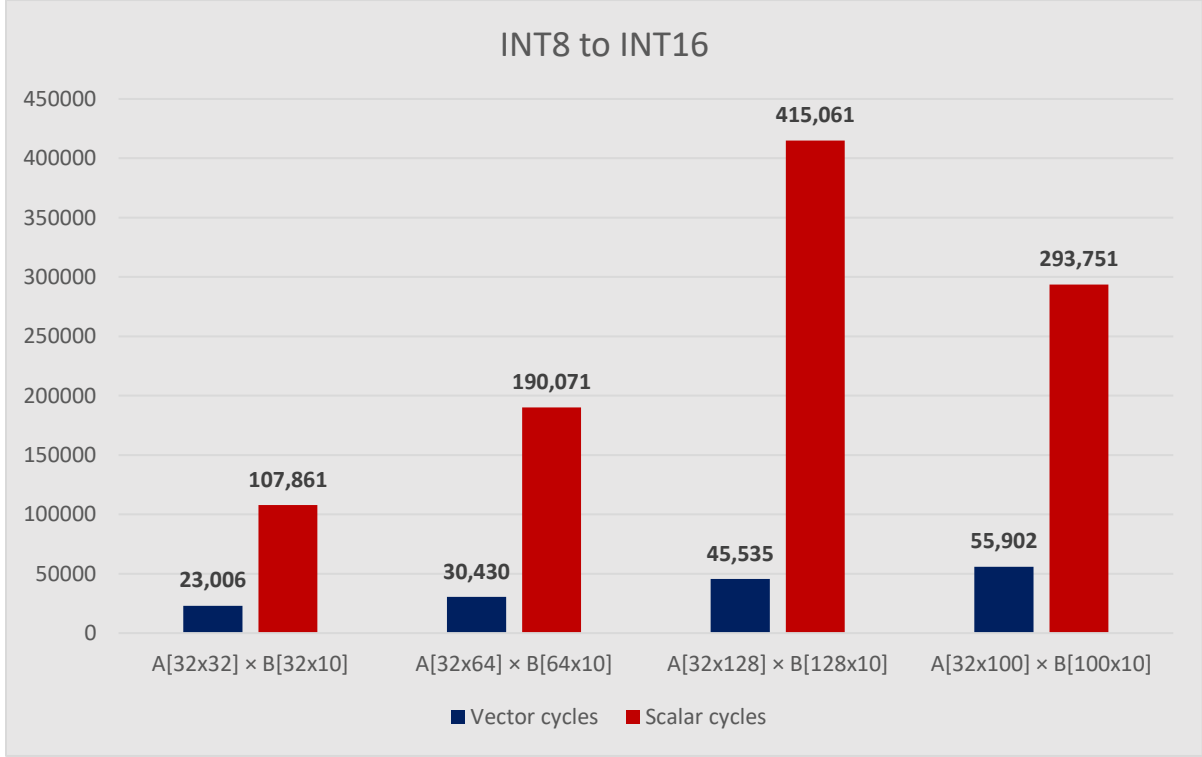



```
em_level/Simulation$ python3 compare_outputs.py --scalar-mem scalar_output.mem -
-vector-mem vector_output.mem --golden-hex golden_output.txt --elem-bytes 1

======================================================================
  VERIFICATION SUMMARY
======================================================================
  scalar vs golden     PASS  (4096/4096 elements match)
  vector vs golden     PASS  (4096/4096 elements match)
  scalar vs vector     PASS  (4096/4096 elements match)
======================================================================

*** OVERALL: ALL CHECKS PASSED (3/3 comparisons computed) ***
```

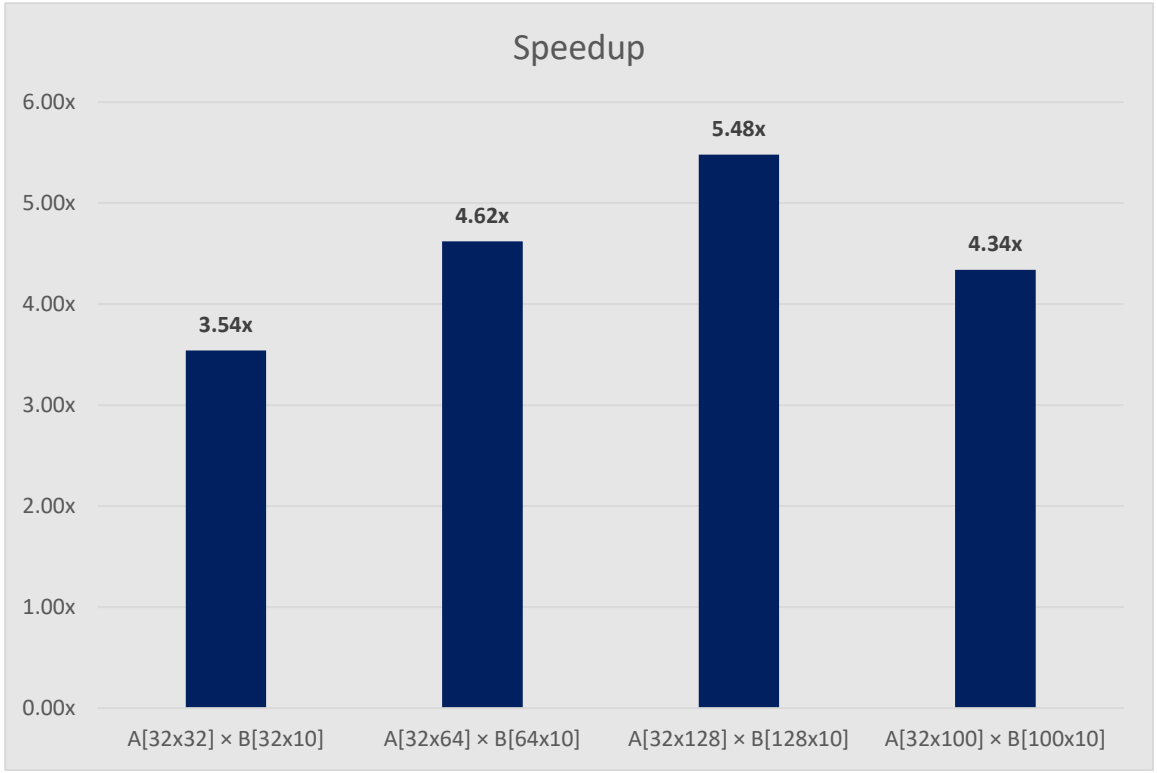
06
Results & Benchmarking

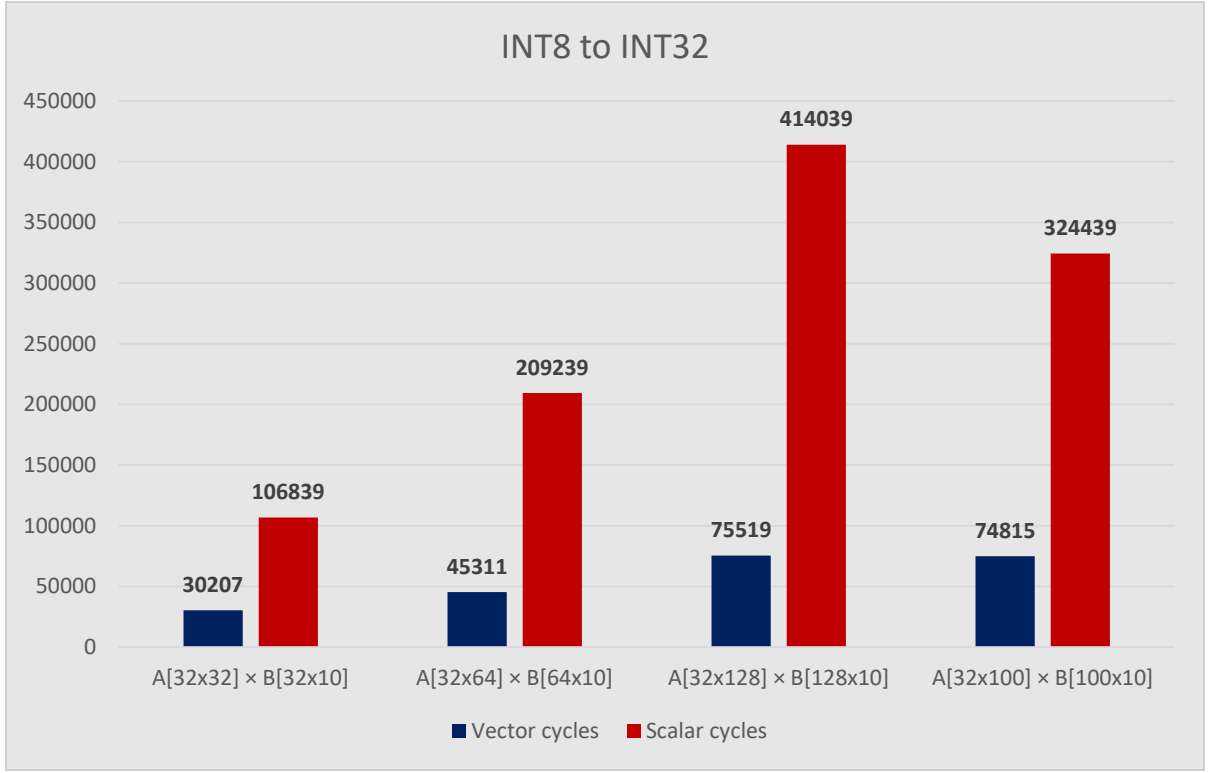
GeMM


Speedup
10.00x
9.00x
8.00x
7.00x
6.00x
5.00x
4.00x
3.00x
2.00x
1.00x
0.00x
4.69x
6.25x
9.12x
5.26x
A[32x32] × B[32x10]
A[32x64] × B[64x10]
A[32x128] × B[128x10]
A[32x100] × B[100x10]

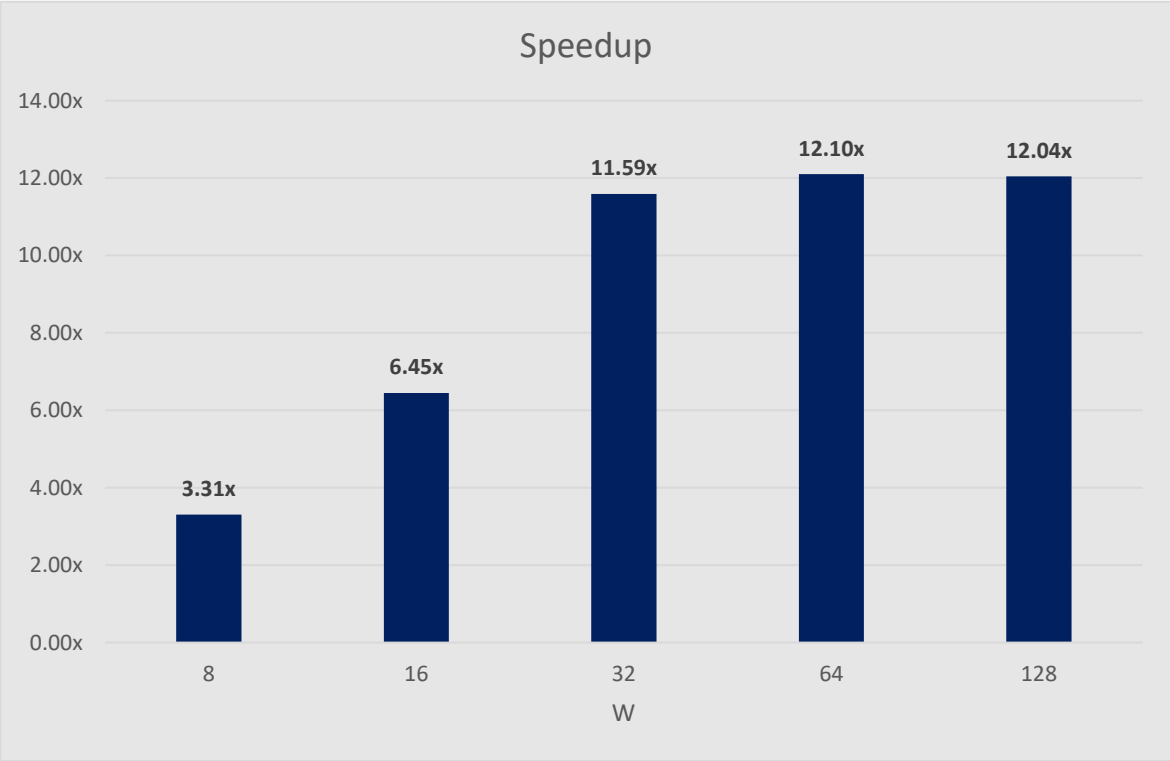

INT8 to INT16
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
23,006
107,861
30,430
190,071
45,535
415,061
55,902
293,751
A[32x32] × B[32x10]
A[32x64] × B[64x10]
A[32x128] × B[128x10]
A[32x100] × B[100x10]
Vector cycles
Scalar cycles

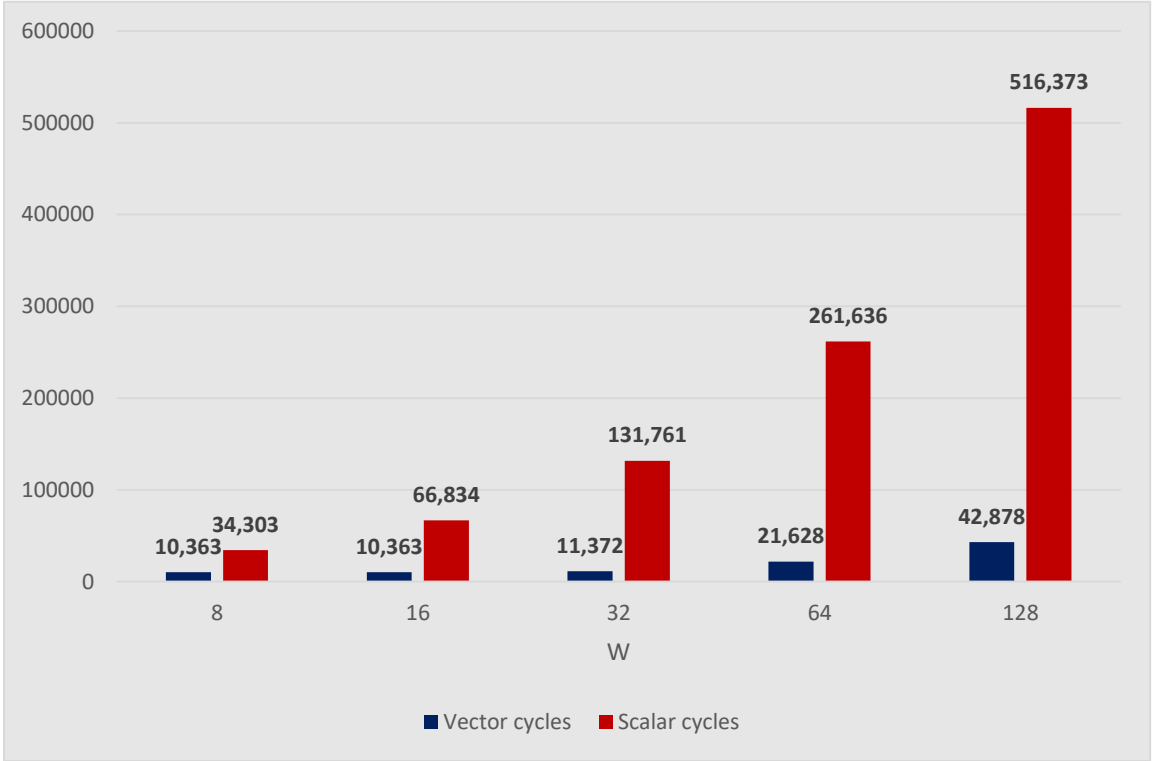
GeMM


Speedup
6.00x
5.00x
4.00x
3.00x
2.00x
1.00x
0.00x
3.54x
4.62x
5.48x
4.34x
A[32x32] × B[32x10]
A[32x64] × B[64x10]
A[32x128] × B[128x10]
A[32x100] × B[100x10]

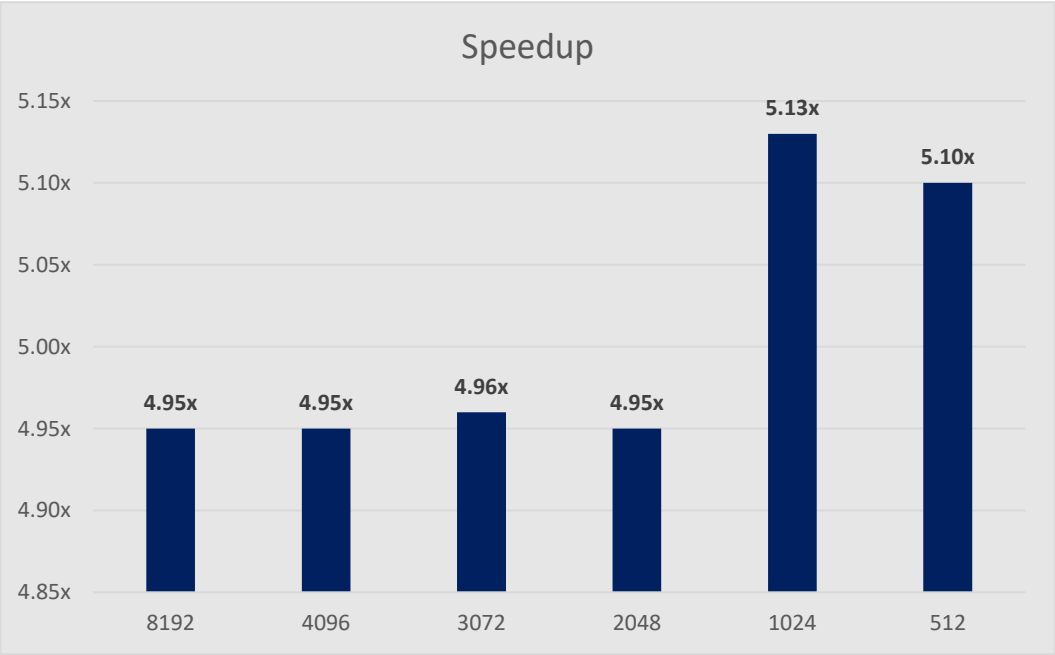

INT8 to INT32
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
30207
106839
45311
209239
75519
414039
74815
324439
A[32x32] × B[32x10]
A[32x64] × B[64x10]
A[32x128] × B[128x10]
A[32x100] × B[100x10]
Vector cycles
Scalar cycles

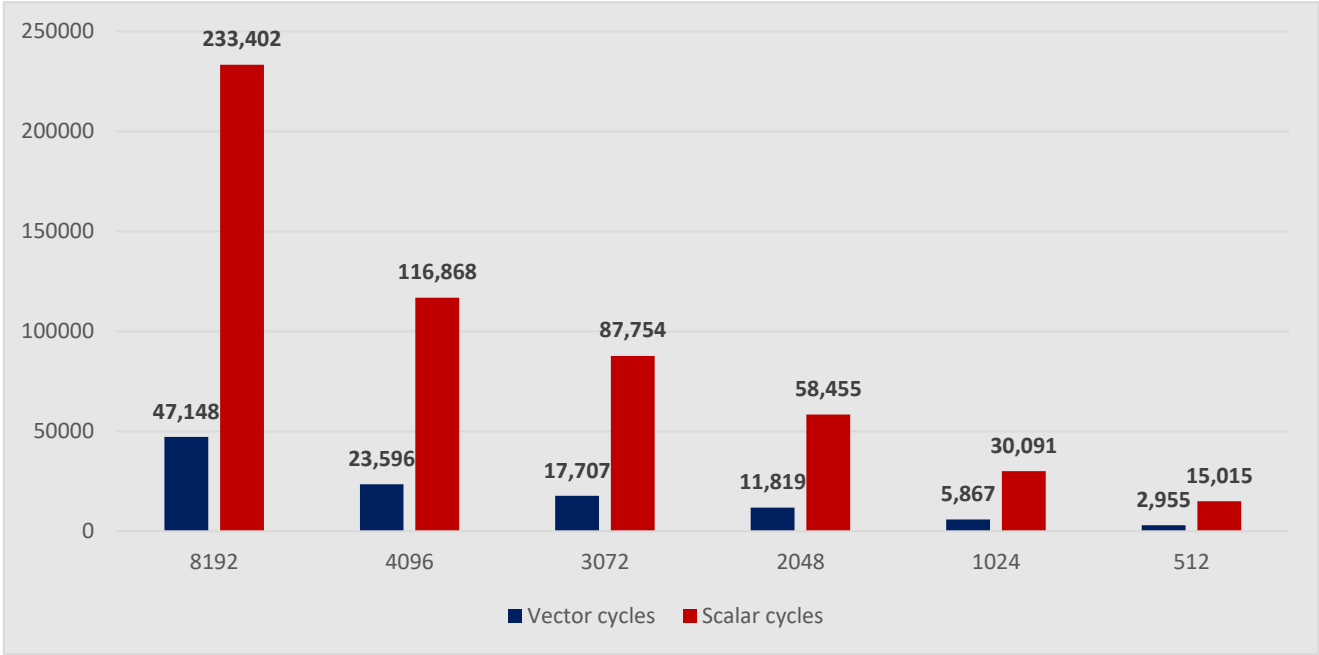
MaxPooling


Speedup
14.00x
12.00x
10.00x
8.00x
6.00x
4.00x
2.00x
0.00x
3.31x
6.45x
11.59x
12.10x
12.04x
8
16
32
64
128
W

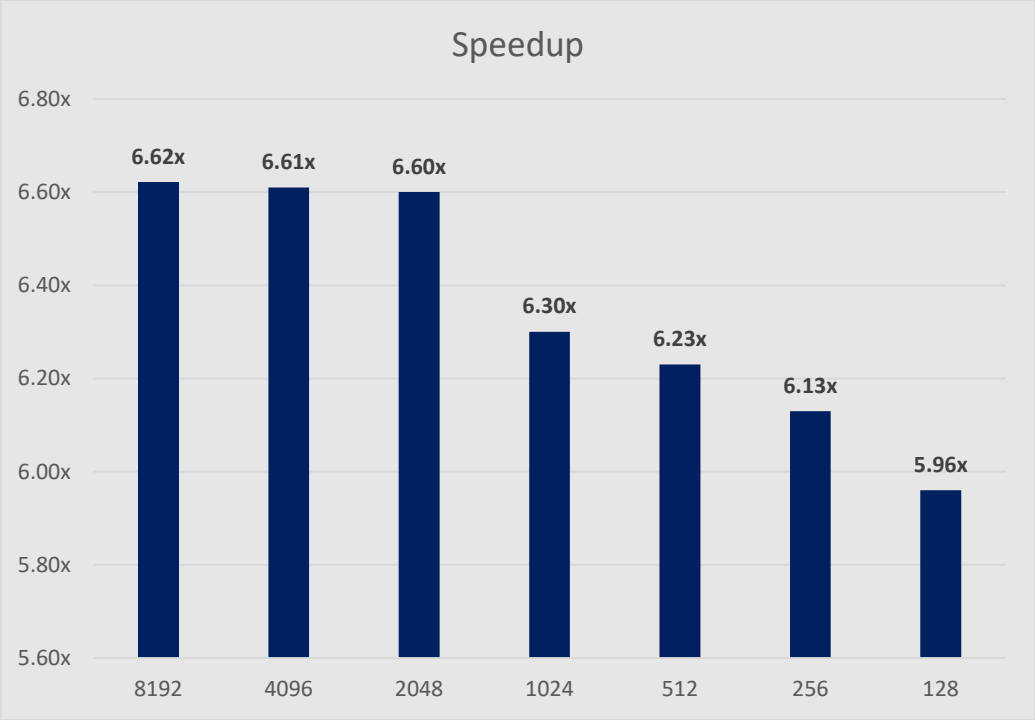

600000
500000
400000
300000
200000
100000
0
10,363
34,303
10,363
66,834
11,372
131,761
21,628
261,636
42,878
516,373
8
16
32
64
128
W
Vector cycles
Scalar cycles

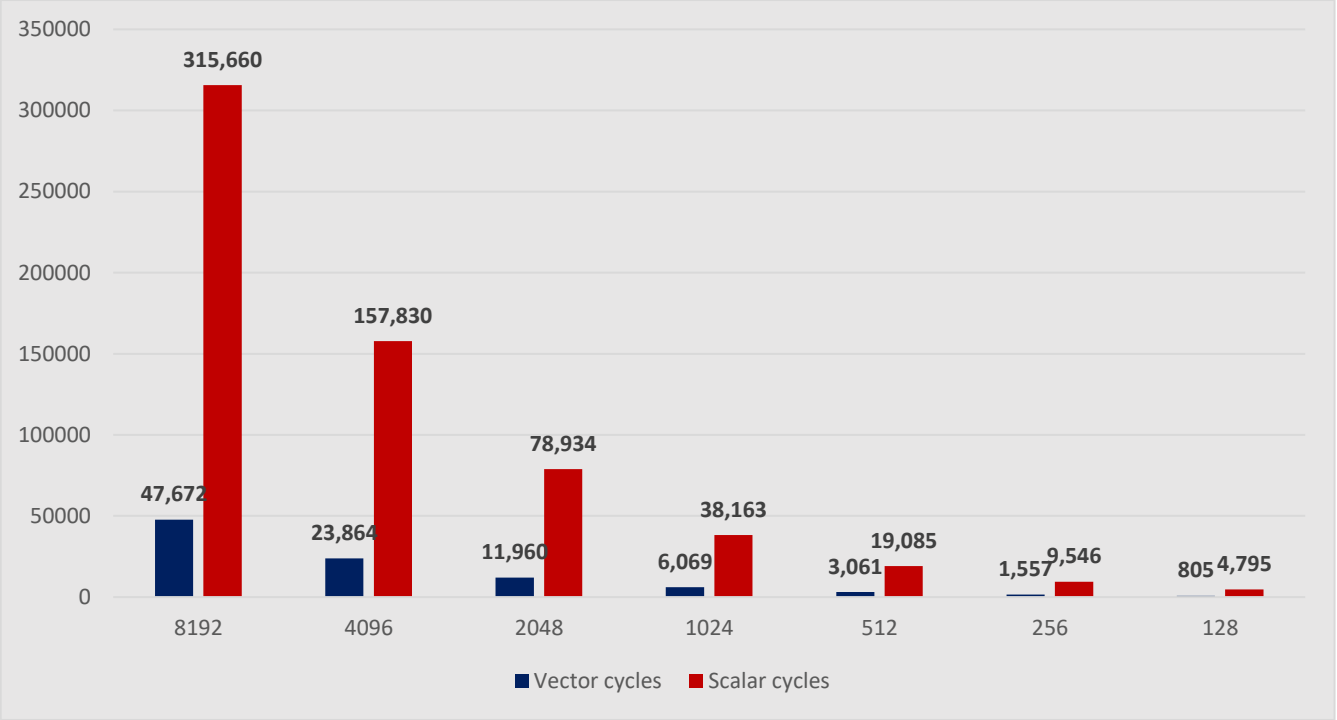
GeLU


Speedup
5.15x
5.10x
5.05x
5.00x
4.95x
4.90x
4.85x
4.95x
4.95x
4.96x
4.95x
5.13x
5.10x
8192
4096
3072
2048
1024
512

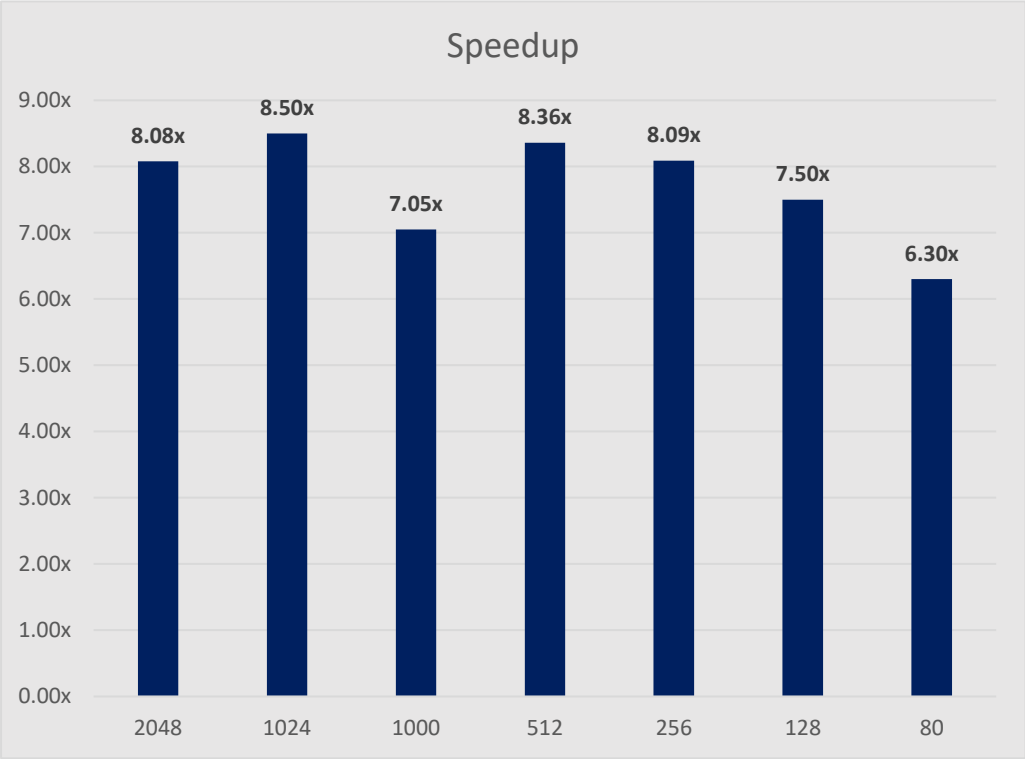

250000
200000
150000
100000
50000
0
47,148
233,402
23,596
116,868
17,707
87,754
11,819
58,455
5,867
30,091
2,955
15,015
8192
4096
3072
2048
1024
512
Vector cycles
Scalar cycles

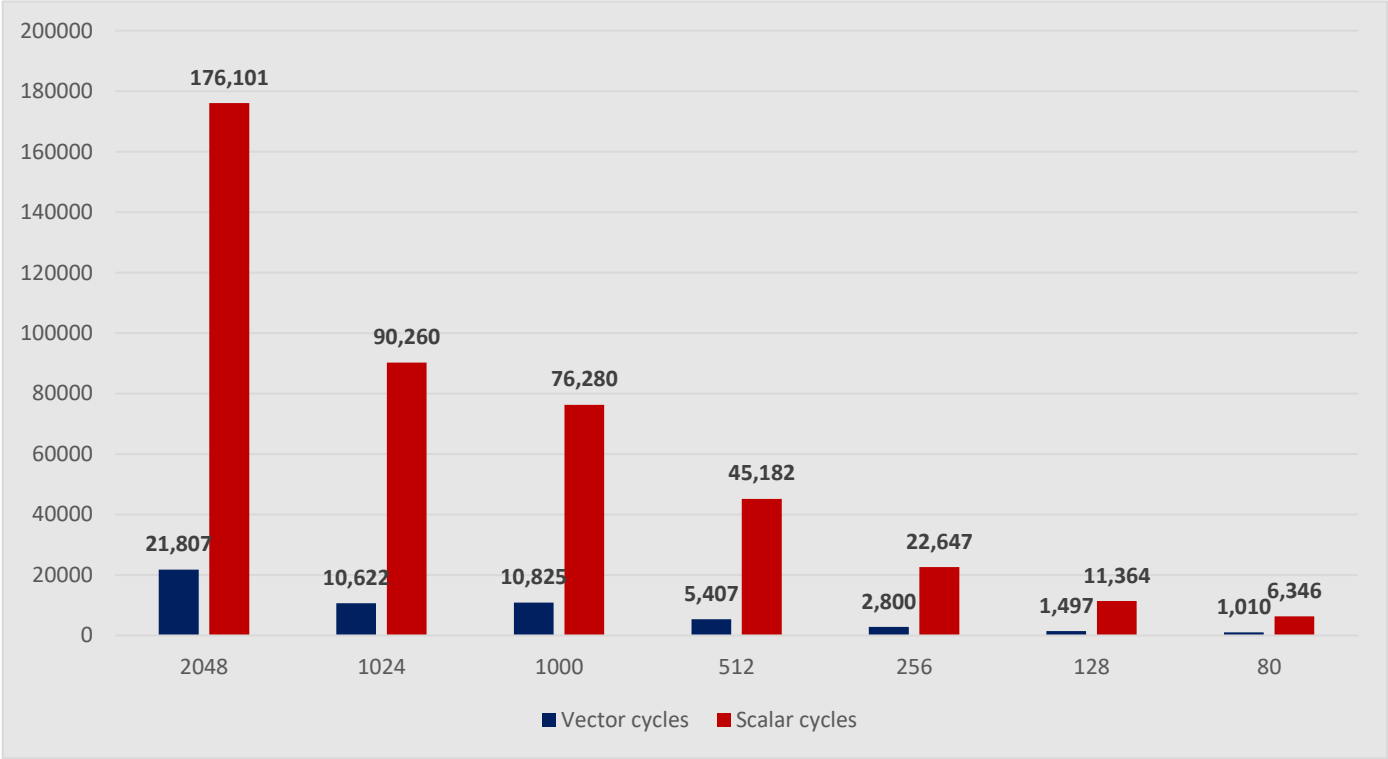
Sigmoid


Speedup
6.80x
6.60x
6.40x
6.20x
6.00x
5.80x
5.60x
6.62x
6.61x
6.60x
6.30x
6.23x
6.13x
5.96x
8192
4096
2048
1024
512
256
128

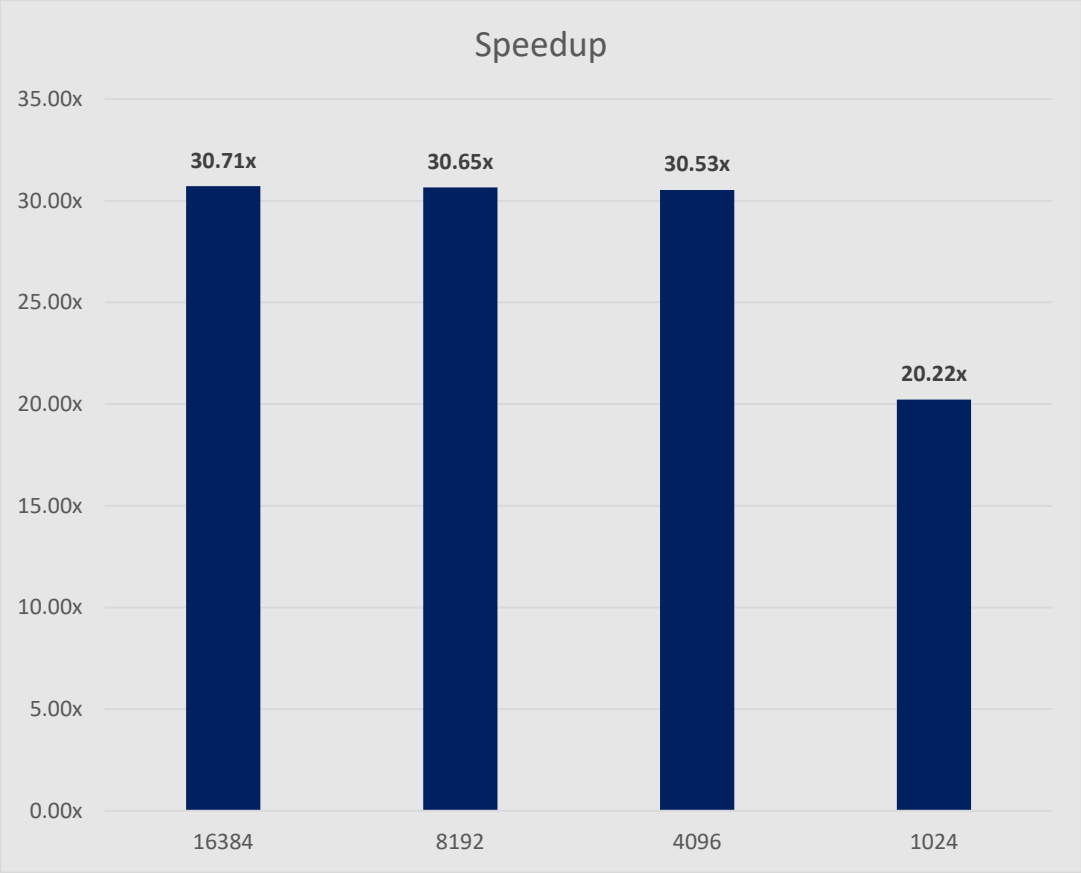

350000
300000
250000
200000
150000
100000
50000
0
315,660
47,672
157,830
23,864
78,934
11,960
38,163
6,069
19,085
3,061
9,546
1,557
4,795
805
8192
4096
2048
1024
512
256
128
Vector cycles
Scalar cycles

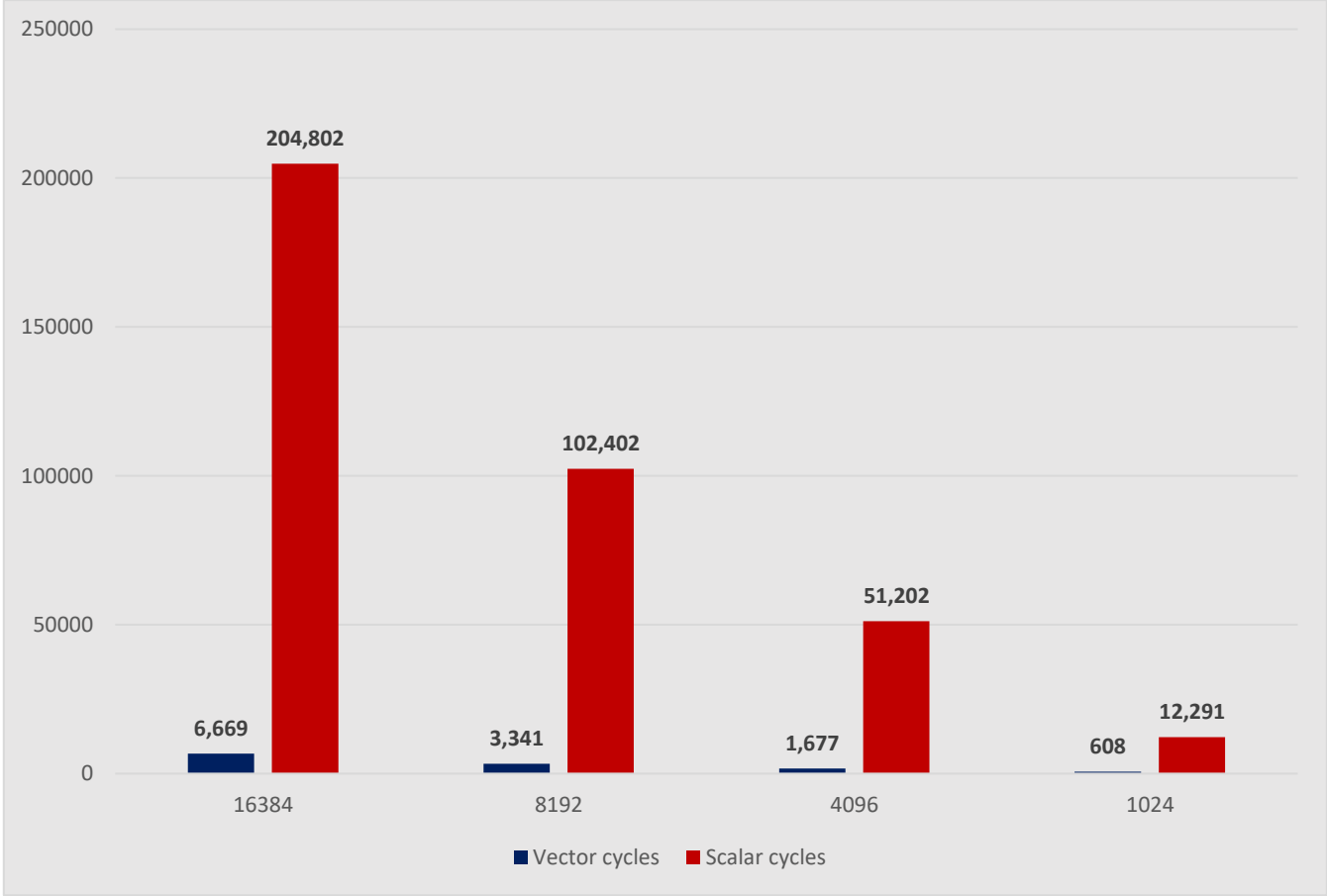
Softmax


Speedup
9.00x
8.00x
7.00x
6.00x
5.00x
4.00x
3.00x
2.00x
1.00x
0.00x
8.08x
8.50x
7.05x
8.36x
8.09x
7.50x
6.30x
2048
1024
1000
512
256
128
80

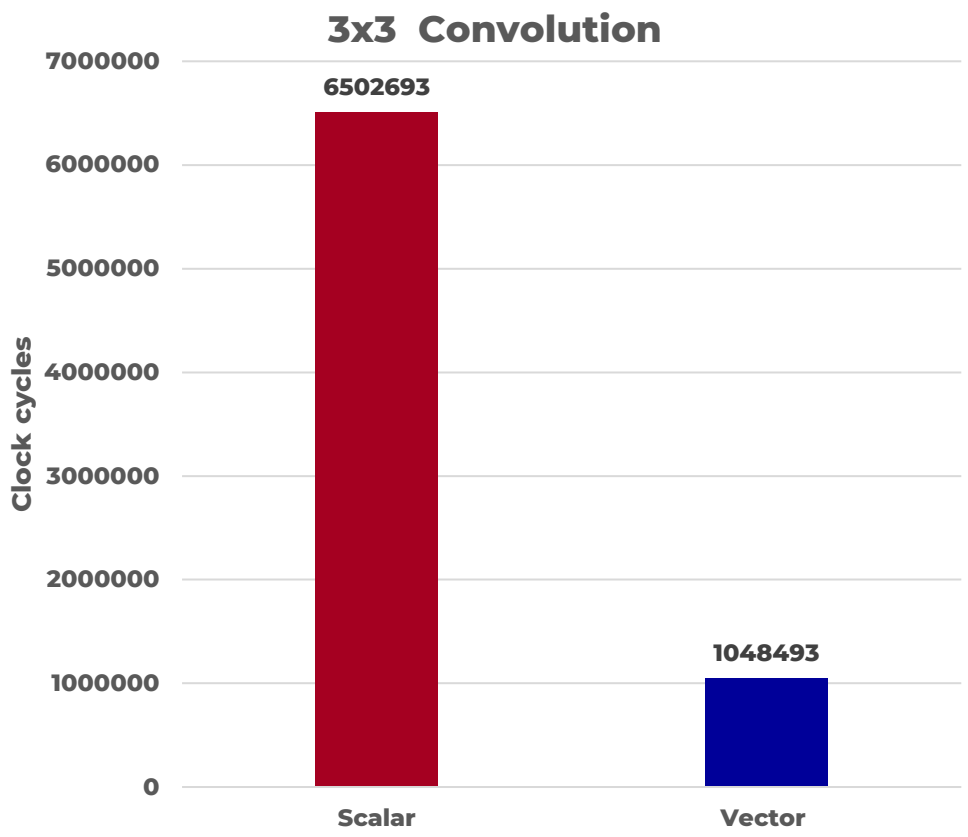

200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
176,101
90,260
76,280
45,182
22,647
11,364
6,346
21,807
10,622
10,825
5,407
2,800
1,497
1,010
2048
1024
1000
512
256
128
80
Vector cycles
Scalar cycles

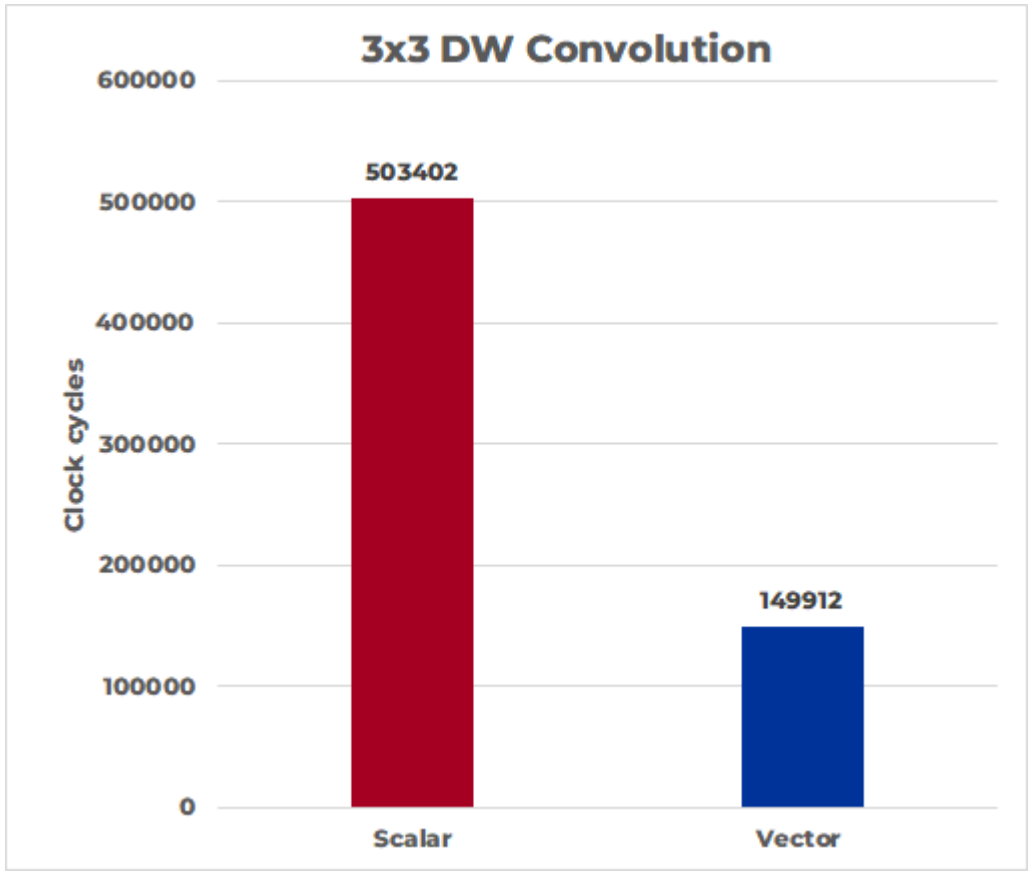
ReLU


Speedup
35.00x
30.00x
25.00x
20.00x
15.00x
10.00x
5.00x
0.00x
30.71x
30.65x
30.53x
20.22x
16384
8192
4096
1024


250000
200000
150000
100000
50000
0
204,802
102,402
51,202
12,291
6,669
3,341
1,677
608
16384
8192
4096
1024
Vector cycles
Scalar cycles

Convolution

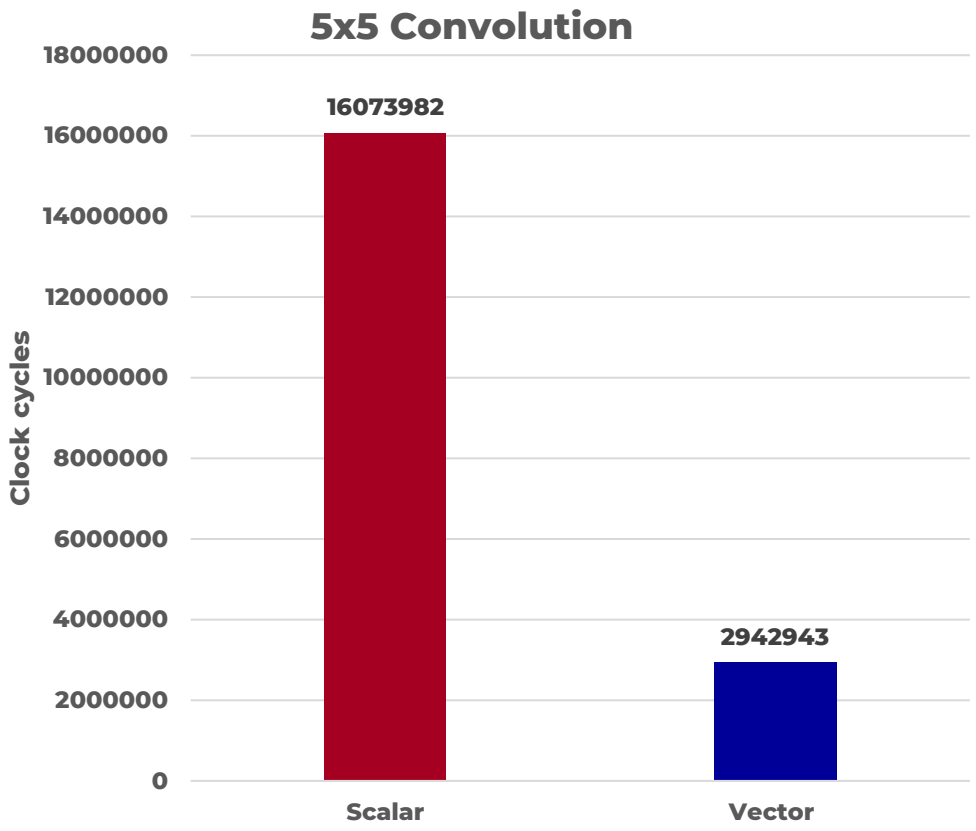


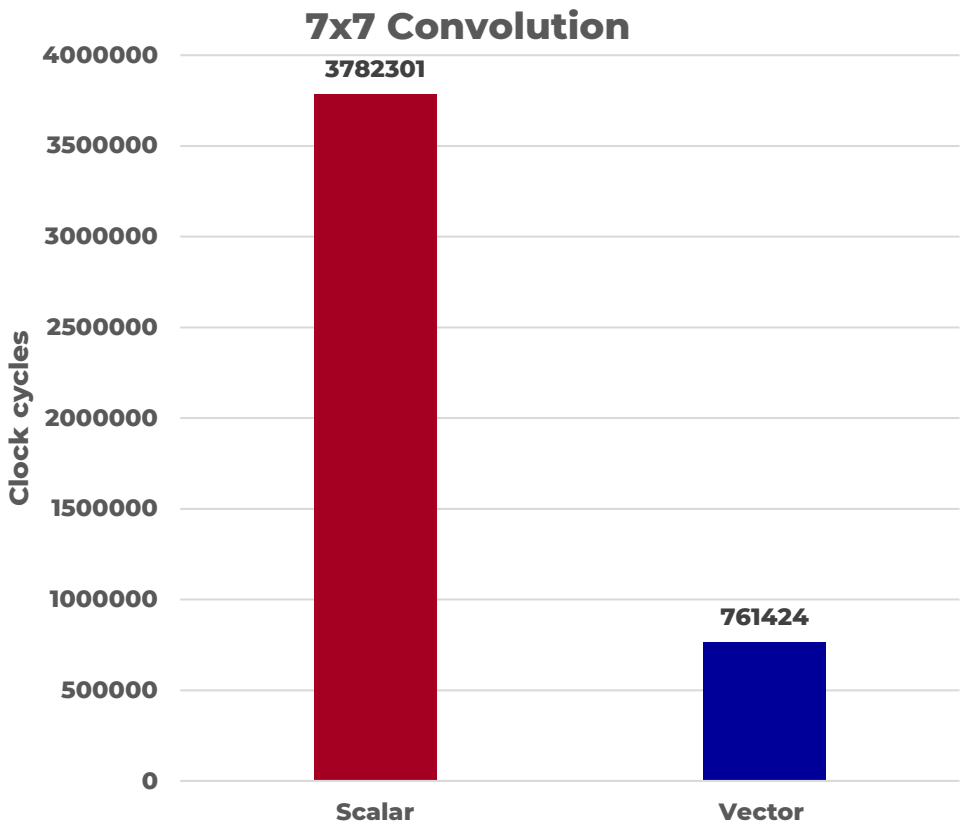


Acceleration: 6.2x

Acceleration: 3.36x

Convolution

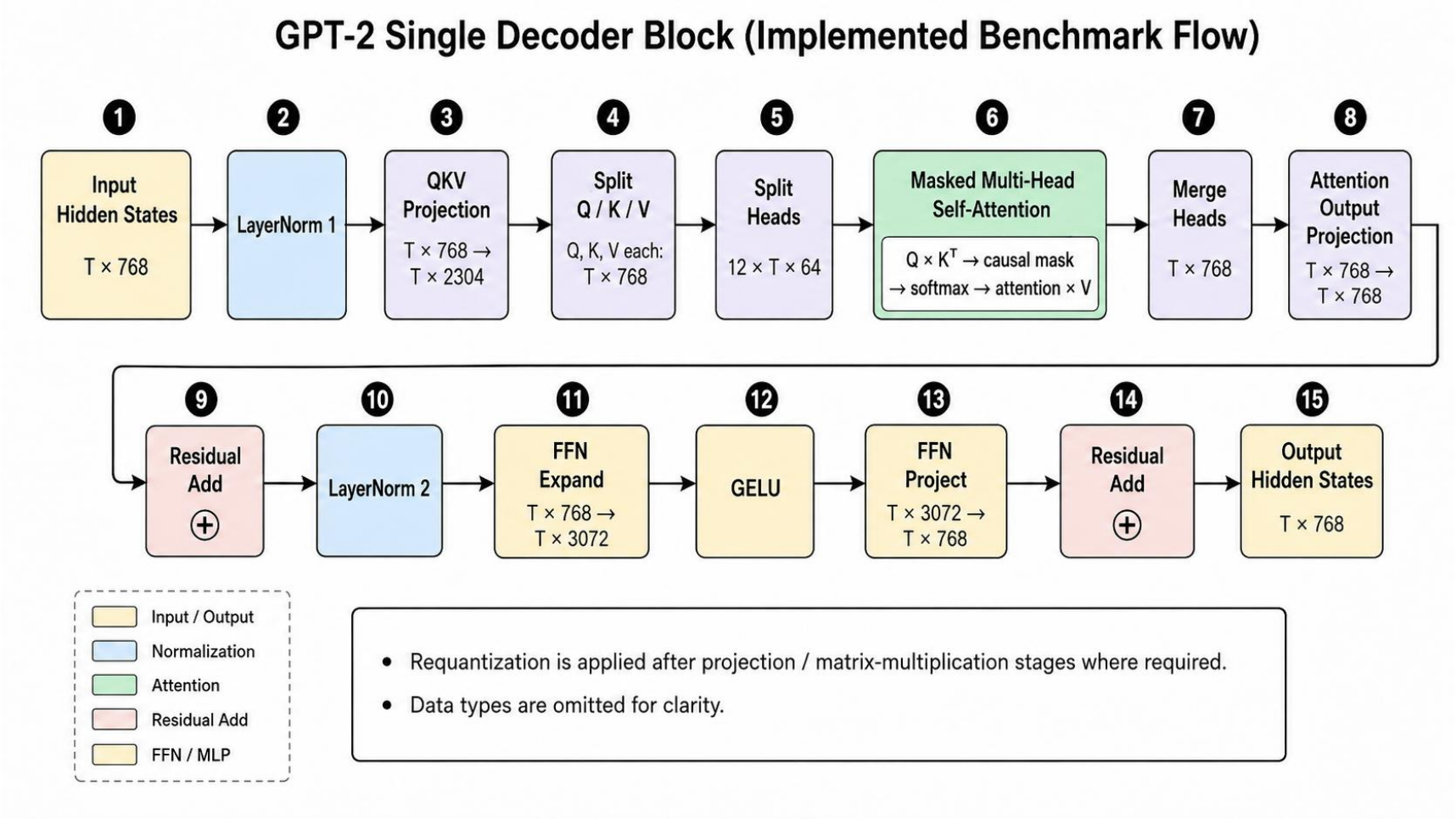




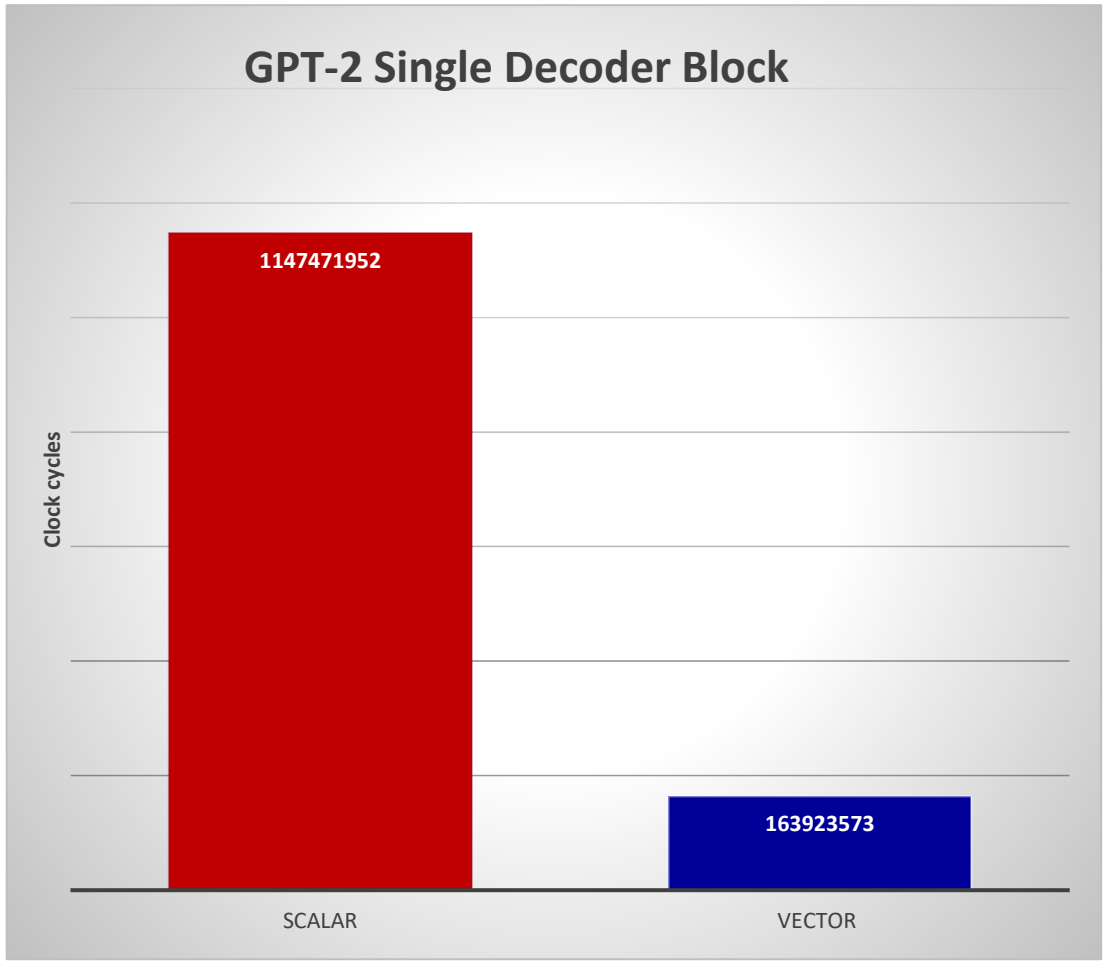
Acceleration: 5.462x

Acceleration: 4.96x

Benchmarking (GPT-2)

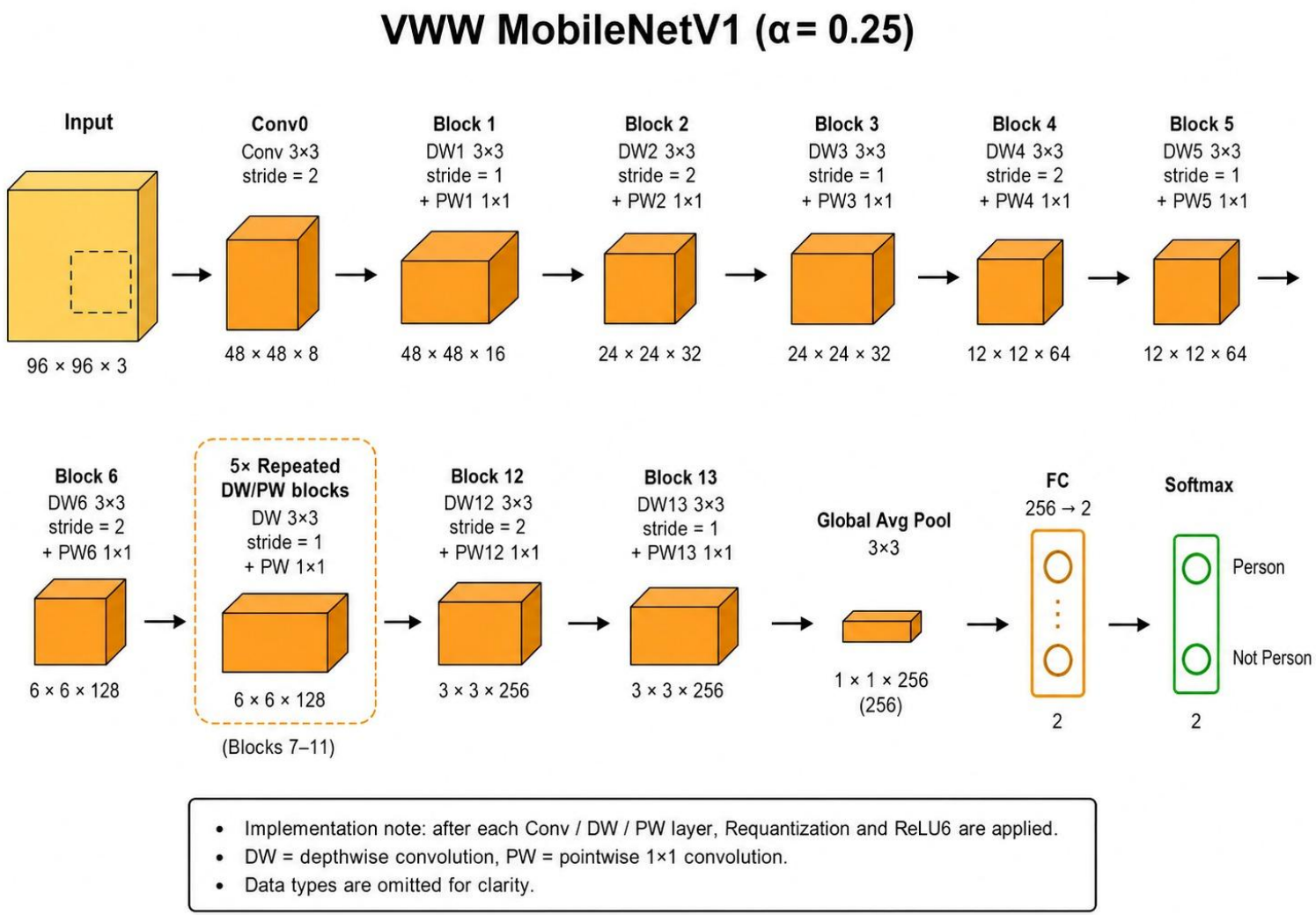


Benchmarking (GPT-2)

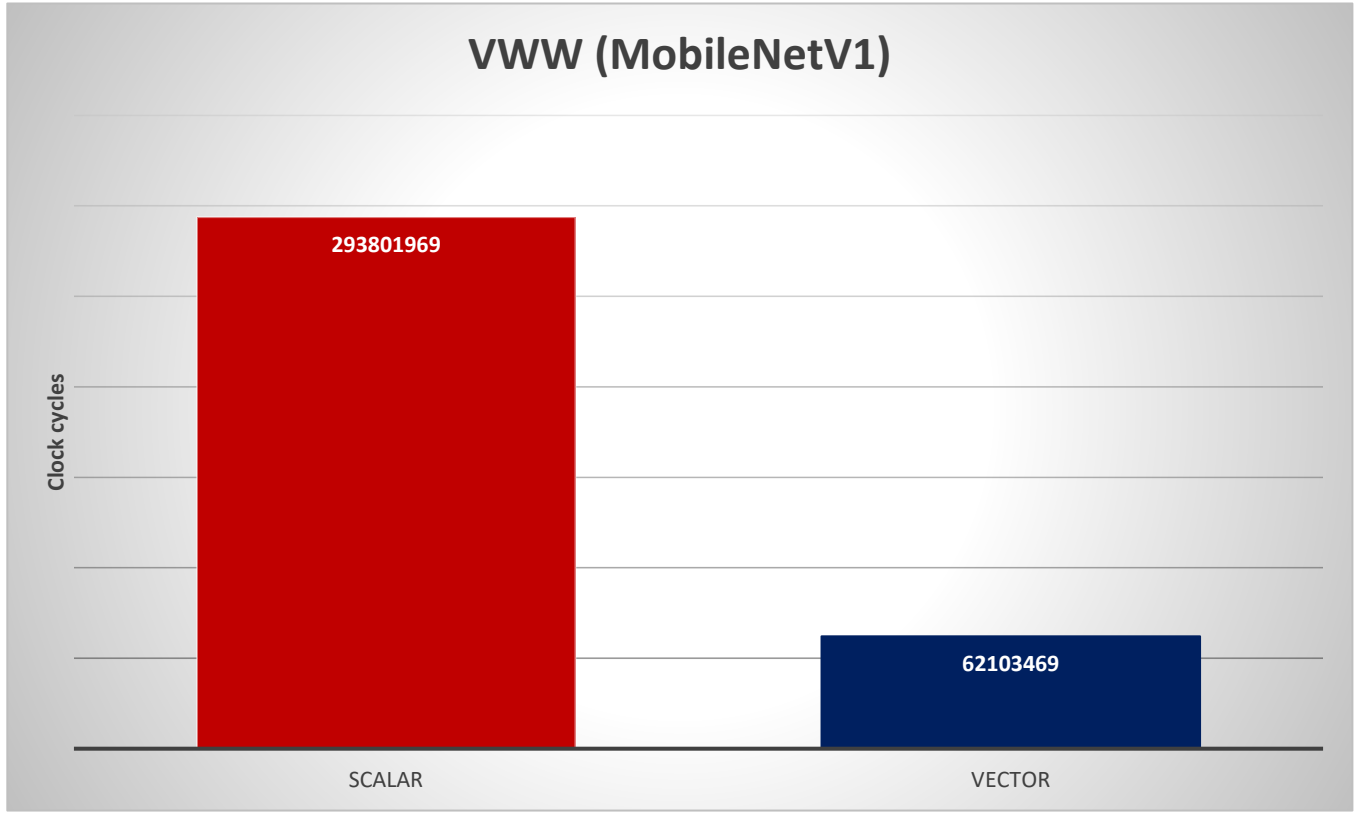


Acceleration: 7x

Benchmarking (VWW MobileNetV1)



Benchmarking (VWW MobileNetV1)



Acceleration: 4.731x

07
Conclusion

Conclusion

07

We designed, built, and verified a custom RISC-V vector extension on the CV32E40P core, then benchmarked it on real CNN and transformer workloads.

4.73×
VWW MobileNetv1

7×
transformer layer
GPT-2 decoder

100%
instructions verified
on the integrated RTL

Edge AI needs compute that runs locally.
This extension is a step toward delivering it on a small, low-power core.

Thank you

Questions & discussion